Bolt Engine Cheatsheet

This document is an UNOFFICIAL concepts and code reference for use with BoltEngine 1.0

This document is not a great ‘tutorial’ resource, but instead should be read after a tutorial to get a better idea of what is really going on. It is also useful if you are familiar with other Unity Networking systems and want to get straight into the meat of how Bolt works. This description focuses on the overall architecture of the Bolt Engine, and how a game would be written using Bolt (including a cookbook section).

This is not an official “Bolt Engine” document. It is being written to help the Bolt Community.

The latest official BOLT TUTORIAL is at https://doc.photonengine.com/en/bolt/current/getting-started/bolt-101-getting-started

Table Of Contents

1. BoltEngine Fundamentals

With Unity Personal all features should be available as of 0.4.3 and Unity 5

The heart of the BoltEngine programming model: The BoltEntity

Bolt makes Unity’s GameObjects into BoltEntities which can sync state over the network automatically, and with lots of opportunity for prioritizing/customizing those sync operations. Unlike other Unity network systems, by default Bolt does the serialization and syncing. Essentially you add code / adjust parameters to change the default serialization/syncing methods.

The main Bolt docs have a good explanation of BoltEntities, Entity ownership, and Entity properties so read that information first.

The key points are that:

- The BoltEntity is a Unity GameObject with the BoltEntity script attached - that ensures the GameObject (and Bolt Entity) will be represented/replicated on the network by Bolt.
- A BoltEntity is defined in the Unity Editor by adding a Bolt Entity script component to a normal Unity Game Object (almost always a prefab)
    - To be precise on nomenclature, 'BoltEntity.cs' is the script used to make a GameObject into a "BoltEntity" as described here.
- A BoltEntity must be instantiated at runtime over the network in one of the following manners:
    - 1) By calling `BoltNetwork.Instantiate()`
        - Using a `BoltPrefabs.<theBoltGeneratedEntityId>` parameter or
        - Using a `Resource.Load(path)` GameObject Parameter (as long as it is a Prefab that contains a BoltEntity).
        - The **Owner** can reliably pass information about the entity to the other nodes as an IProtocolToken at this time, and they can read this when they `Attach()`
    - 2) By having an object in the scene which has a Bolt Entity.
        - Note that there are issues with this in 4.1.x - see this forum discussion.
        - The owner is the host
- The content in the shared/replicated information is defined using a Bolt State in the Unity Editor (Window -> Bolt Engine -> Bolt Assets)
    - The State can have multiple parameters. Each parameter can have individual choices for replication scope, priority etc.
- The request of changes to State is as follows (note that is uses some terms defined later in this doc, but is here for completeness):
    - The entity's **Controller** node has a callback called `SimulateController()` and this puts the ***requested*** actions (move, fire etc) into a Bolt Command.input structure.
    - Bolt then sends this Command to the **Owner** (the node which is the authoritative server for this entity)
    - The entity's **Owner** node decides what is valid to do, then updates the entity's State object properties in a callback called `ExecuteCommand()`.
    - Bolt then replicates the **Owner**'s new State to the entity's **Proxies** and (optionally, depending on a per-property flag in the State definition) the **Controller**.
    - There is also an optional step where Bolt sends the 'result' back to the **Controller's** `ExecuteCommand()` with a Command with resetState=true and the decided values in Command.result - this is used for Authoritative servers typically.

Bolt will destroy these entities in the following cases only:

1. If you disable the GameObject the BoltEntity script is on
2. If you load a scene and the GameObject is not marked as "don't destroy on load"
3. ??Disconnect??

The key purpose of Bolt is this abstraction it provides: efficiently syncing the State of BoltEntities (as a result of Commands sent from **Controllers** to **Owners**). In particular, it has an efficient wire protocol - it compresses the State changes into a byte array and sends diffs over the network to remote nodes - and has prioritization options for the network traffic.

Bolt very naturally supports the mode where one of the player nodes is the server. However, it has no way yet to migrate/failover state to a new server (e.g. as uLink does).

The BoltEngine Developer experience

Bolt installs additional Editor experiences into Unity for working with Bolt and Bolt Assets, and menu options to recompile generated code.

Normal Unity Prefabs are made 'special' by adding a BoltEntity component to them.

- **After this, you **must** run 'Assets -> Bolt Engine -> Update Prefabs Database in order to continue the Bolt workflow.**

Bolt States & Commands are used to define how replication works for the properties on a Bolt Entity. They are defined in the Bolt Editor Window. Use the Bolt Assets window (Window -> Bolt Engine -> Bolt Assets) to view and create States and Commands.

- **After this, you **must** run 'Assets -> Bolt Engine -> Compile Assembly' in order to continue the Bolt workflow.** This auto-generates the code required to sync the Bolt Entities based on your definitions.
- There are some additional Bolt assets which will be discussed later in this doc - Events and Structs (now called Objects). These are defined and generated the same way.

BoltEngine Debugging and Logging workflows

- Bolt has some workflows that are nicer on Unity Pro ("Window -> Bolt Windows -> Scenes -> "Debug Start" scene -- This starts clients & server), but Bolt overall works fine with the free Unity version.
- As of Bolt 0.4.1.x there are DEBUG builds of Bolt (you decide to have them by installing the debug version of Bolt) which do a lot of checking for coding mistakes, but are 4x-5x slower than the RELEASE builds. See http://doc.photonengine.com/en/bolt/current/setup/builds for details.
- There is also a **separate** "debug mode" in Bolt Settings - this shows information about the Bolt Entity that you are looking at.

Logging

- There is also a Bolt debug 'console' that can be toggled and/or shown by default. See Bolt Settings for the toggle and initial state. The toggle defaults to TAB.
    - It is possible to save the console output to a file - see Bolt Settings; "log targets" -> (select) "file". This creates a timestamped file, such as `Bolt_Log_SERVER_2014Y-12M-29D_21H8M48S_287MS.txt`
    - If you want it intertwined with the unity player log, you can enable the player log in the unity player settings, and then "log targets" -> "unity"
    - You can optionally have the Unity `Debug.Log()` output be shown in the Bolt debug console.
    - It's not really a 'console' - it doesn't allow commands to be edited/input
- There are BoltLog.Info(), .Warn etc methods that use this debug terminalA handy wrapper for `BoltLog.Info()` in Entity-related classes is the following - it will show the class, entity name and entity Role (**O**wner / **C**ontroller):

```
private void _tracer(string msg)
{
    if (fNoisyDbg)              // Define this somewhere in the Class. Set it in Inspector or programmatically when you want noise
```

```
            BoltLog.Info(this.GetType().Name + ".cs:" + entity.name + "[" + (entity.isOwner ? "O" : "") + (entity.hasControl ? "C" : "") +
"]: " + msg);
        }
```

BoltEngine and version control

- The special Bolt Assets - i.e. Bolt Commands, Bolt Events, Bolt States, Bolt Objects are defined in the editor but stored in `Assets/bolt/project.bytes` so this file should be committed when you change these. As the name implies, this is a binary format at present :(
- When updating to a new version of Bolt, it is best to then commit the bolt dll in the project - this is at `Assets/bolt/assemblies/bolt.dll` and `Assets/bolt/assemblies/bolt.user.dll`

Upgrading Bolt versions

- The upgrade process in Bolt is quite simple. An official upgrade manual can be found at: http://doc.photonengine.com/en/bolt/current/setup/upgrading

BoltEngine supported platforms & their quirks (TODO)

Target	Client constraints	Server constraints	Debugging notes
Standalone player for Mac/PC			
iOS			
Android			
Windows Store Apps	Not supported yet		
WebGL	Not supported yet	Be aware of crossdomain.xml requirements	
PS Vita	Not supported yet		
XB1	Contact support@boltengine.com for info		
PS4	Contact support@boltengine.com for info		
Steam	Contact support@boltengine.com for info		

2. The three main Concept Groups in Bolt

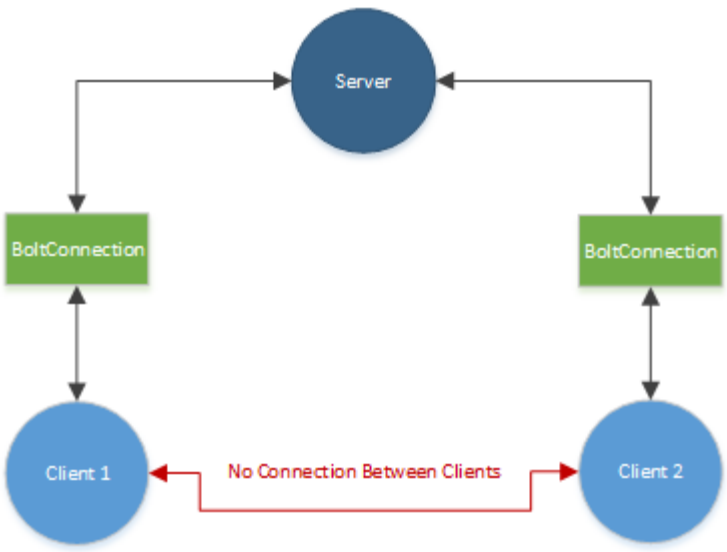
Bolt has three related but different groups of concepts - network roles, entity roles and message types. **Do not mix these concept groups up or you will have a bad day.** They are interrelated but different.. for example the *Server* might be an **Owner** for some Entities, but not for others..

The tables below summarize the key aspects of these concept groups and their concepts:

Concept Group	Purpose	Sub-concepts
Bolt Network Roles	Network Roles determine which of the nodes is the (single) *Server*, and how the nodes are connected to the Server.	*Server* and *Client*
Bolt Entity Roles	Entity Roles determine (for a given entity) which node will own that Entity's state, which node can request changes to that State (via Commands), and which nodes can only receive State changes	**Owner**, **Controller** and **Proxy**
Bolt Message types	Define the kinds of messages that are sent between Bolt nodes in order to communicate state change requests or notifications.	Commands, States and Events

In this document we define 'node' as being an instance of a Unity runtime that is using BoltEngine and intends to connect with other nodes to enable multi-player gaming.

Bolt Network Roles (*Server* and *Client*)



A node must have exactly one Network role: either a *Client or a Server.*

- *Clients* connect to *Servers* via *Connections.*
- Currently, in Bolt 0.4.1.x, there are no connections between *Servers*, and no connections between *Clients*. - i.e. it is a hub-and-spoke topology only. That may change when Master Servers come in

Bolt is using what is commonly referred to as the 'Client/Server Model', this means that one computer is considered the *Server*, everyone else is considered a *Client* and they connect directly to the *Server*. There are a couple of things to note about this.

1. The server have one BoltConnection object *per client*. All client connections can be found under BoltNetwork.clients. On the clients themselves this will return zero items.
2. The clients have *only one* BoltConnection which is the connection to the *server*. This connection can be found at BoltNetwork.server. This property returns null on the server itself.
3. There is no connection that refers to *yourself* in any way, they always represents a link to *another* computer.
4. There are no direct connections between *clients*, all data is passed through the *server*.

Bolt Network Roles:	*Server*	*Client*
What is the job of a node that has this Network Role?	The *Server* is the node that is hosting the game. If the developer wants an "Authoritative Server" then the *Server* will also be the **Owner** of (most) BoltEntities in order to prevent cheating and ensure that there is a consistent game view (see next section).	A *Client* is normally any node that is connected to the game but is not considered to be the *Server*.
How many nodes can have this Network role?	Exactly 1	0 or more
Can a node have additional Network roles?	No	No
How is this Network Role assigned?	Node selects Network role during startup (See below)	Every remote connection from the *Server* is a *Client*.
Code example of assigning this Network Role	`BoltInit.cs` has special Bolt Internal code that handles *Client* and *Server* initialization using the `BoltLauncher` class. Whilst starting to use Bolt, `BoltInit.cs` should be sufficient. For clues on how to roll your own Client/Server initialization for advanced/customer scenarios, see this post from fholm that describes `BoltNetwork.InitializeServer()` and `BoltNetwork.InitializeClient()` which you can call directly. You can check in the BoltDebugStart.cs and BoltDebugStartNonPro.cs scripts for more clues.	
Code example of checking if a node has this Network Role:	`(BoltNetwork.isServer)`	`(!BoltNetwork.isServer)`

Specific callbacks the node will get ONLY if it has this Network Role	`void ConnectRequest(endpoint, token);`
	`void Connected(connection, token);`
	`void ConnectFailed(endpoint);`
	`void ConnectRefused(endpoint);`
	`void Disconnected(connection);`
		????

Disconnect	`BoltLauncher.Shutdown();`	`BoltLauncher.Shutdown();  // Disconnect from Server`

Connections act as a conduit for transferring messages between nodes. Within Bolt, a connection is considered to be a link to another *Client*/*Server* over the network. This means a connection can refer to either a *Client* or the *Server*.

Note that there are settings for enabling manual connection acceptance in the BoltSettings window. Once enabled, the Server can have code to accept/reject connections, such as:

```
public override void ConnectRequest(UdpKit.UdpEndPoint endPoint, IProtocolToken token)
{
  if (/* some test */)
    PlayerData data = (PlayerData) token;
  else
    BoltNetwork.Reject( endPoint);
  // etc...
}
```

0.4.3 no longer need override?

// TODO - also note what Sessions are (src).. Scope modes and maybe Scene IDs.

Bolt Entity Roles (Owner, Controller and Proxy)

Each individual instantiation of a Bolt Entity has a well-defined **Owner**, **Controller**, and **Proxy/Proxies.** This Entity Role determines whether the entry initiates, validates/applies, or receives state changes.

In a nutshell, The **Owner** 'owns' the State of the object and can replicate it out to other nodes. The **Controller** has the right to send Commands to the **Owner** requesting State changes. The **Proxy** just receives state changes.

In more detail, the roles are as follows:

Bolt Entity Roles:	**Owner**	**Controller**	**Proxy**
What is the job of a node that has this Entity Role for a specific instantiated Bolt Entity? See also http://doc.photonengine.com/en/bolt/current/in-depth/entity-ownership	Causes a BoltEntity to be instantiated using BoltNetwork.Instantiate() and then implicitly becomes the Owner for this entity. Owner applies self-initiated State changes, and/or requested State changes (**Owner** gets these requests from the **Controller** as Commands and if valid, applies them to the State during `ExecuteCommand()`) The **Owner** of a BoltEntity distributes the resulting state changes to **Proxies**. The Owner will also optionally send this to the **Controller** (if the Joystick icon in the State's property in the Bolt Editor is enabled). (src) The State of a BoltEntity can only be modified by the **Owner**. Trying to change the State from a **non-Owner** node will mean the State will not get replicated over the network.	Sends State change Requests to the Bolt as Commands to the Entity **Owner's** command queue - for example a movement intent based on player input. A **Controller** cannot change a Bolt Entity's State directly. Any State changes it makes directly will **not** be replicated to **Proxies**/**Owner**. Similarly, "having" **Control** of an entity does **not** mean you can move it directly with a transform; you still need to use commands - i.e. create a Command in `SimulateController()` and execute it in `ExecuteCommand()`. This will of course work locally, and will replicate over the network if your entity happens to also be the **Owner**, but that is usually an 'accident' of coincidence when you happen to test using a Player on the *'Server'* node in a p2p-style setup with an Authoritative Server.	ONLY show State changes (Gets these State changes from **Owner**) Cannot initiate state change requests to **Owner** because `SimulateController()` does not get called since node is not a **Controller** for this Entity. Updates to State would not get replicated since it is not an **Owner**.
How many nodes can have this Entity Role for a specific instantiated BoltEntity	Exactly 1	0 or 1	0 or more
Can a node have other Entity Roles also?	Yes: **Owner** might also be **Controller** (e.g. Player on a p2p 'server')	Yes: **Owner** might also be **Controller** (e.g. Player on a p2p 'server')	No.
How is the role node determined	The Instantiator of the Entity becomes the owner by default, unless specifies who the owner will be. Ownership cannot be transferred.	By default there is no **Controller**. The **Owner** may decide which node will be the **Controller.**	If the node is neither the **Owner** or **Controller**, then it is a **Proxy**

Code example of assigning this role for a specific Bolt Entity 'entity'	Ownership is assigned at create time using `BoltNetwork.Instantiate()`. This can be on the local node (no param), or remotely by specifying the connection of the node that we wish to be the owner
	
	TODO: Example of how to create entity on a connection -- Thus ownership can be set by instantiating on the required connection
		The **Owner** can take control with:
		`entity.TakeControl()`
		
		The owner can assigned control to another node with:
		`entity.AssignControl(connection)`
		
		The most reliable callback to have called once Control has been assigned is:
		`public void override ControlGained() { .. }`
		
		Control can be released with:
		`ReleaseControl(token)` or `RevokeControl(token)`
			n/a - is **Proxy** by default

Test if the node has this role for a specific Bolt Entity 'entity'	`entity.isOwner`	me: `entity.hasControl` other: `entity.IsController(connection)` However - **the Controller role is not initially set at entity creation time** - it is assigned and can be checked this way only once `TakeControl()` or `AssignControl()` has been called. So callbacks like Startup(), Awake() and even Attached() (especially on the **Owner**) are unreliable places for code that should have special behavior when a node is assigned control for an entity. Instead, use `ControlGained()` for this situation.	`(!entity.hasControl && !entity.isOwner)`

Specific callbacks the node will get ONLY if it has this Entity Role	`myController : Bolt.EntityEventListener<iMySerializer>`
	• `BoltEntity.SimulateOwner()`
	  ○ Called for **Owner**, doesn't seem to do much.
	• `BoltEntity.ExecuteCommand(...)`
	  ○ Called also for **Controller**, but only **Owner** always gets invoked
		`myController : Bolt.EntityEventListener<iMySerializer>`
		• `Attached() // binds serializer`
		  ○ Called (once?) after Control has been assigned to this node
		• `BoltEntity.SimulateController()`
		  ○ Generate a Command to send to **Owner**
		• `BoltEntity.ExecuteCommand(...)`
			`myController : Bolt.EntityEventListener<iMySerializer>`
			• ~~`SimulateProxy()`~~
			  ○ Removed from Bolt 0.4.x.x. See notes on why and how to roll your own
			•

	with `ResetState=false`.	○ Called also for **Owner**, but only **Controller** can get invoked with `ResetState=true`.	

Owner/Controller duality is a key source of confusion

Bolt supports p2p games where a *Server* can also be a Player. In that case, the *Server* node has a Player BoltEntity that is acting as both the **Owner** and **Controller** for that Entity. There are a number of special shortcuts that Bolt makes in these cases that do not work in the general case. For example, if a node is both **Owner** and **Controller** for an entity, then if you update State in your `SimulateController()` callback then Bolt will replicate the State changes to other nodes only because that node happens to also be an **Owner**. Be **very** aware of this issue when coding and debugging problems.

Bolt Message types (Commands, States and Events)

The third important related, but different set of concepts are the Bolt Communication types of Commands, States and Events. These are the types of messages that Bolt sends over the network connections between nodes.

Bolt Message Type:	Commands	States	Events
Purpose of Message type	Send intended state change (request) from **Controller** to **Owner.** The message flow also causes these Commands to be sent (directly or back) to the **Controller's** `ExecuteCommand()` callback - details below	Distribute actual state from **Owner** to **Controller** and **Proxies**	Events are the Bolt equivalent of RPC - except there is no return value (somewhat like 'rpc' in uLink) Global events are usually for things like "meta information" around the game, starting a round, game over, but can also be used to communicate inventory for players, etc. Entity events are for quick one-off things like triggering a visual for "explosion" or "took damage", etc. TODO… more.. reference this.
How payload is defined	In Unity Editor "Window / Bolt / Assets" **You MUST recompile Bolt Assets (assembly) after creating/changing a Command in order for the change to have effect.**	In Unity Editor "Window / Bolt / Assets". **You MUST recompile Bolt Assets (assembly) after creating/changing a State in order for the change to have effect.**	In Unity Editor "Window / Bolt / Assets". Note that Bolt intentionally disallows passing large structures like arrays or objects due to abuse of this causing bandwidth issues. Instead the model is to send events describing the delta (i.e consume item), rather than sending full refreshes of the inventory list. **You MUST recompile Bolt Assets (assembly) after creating/changing a Event in order for the change to have effect.**
Destinations See also http://doc.photonengine.com/en/bolt/current/in-depth/replication-modes	Commands are sent by the **Controller** to the local **Controller** and to the **Owner**. The **Owner** may also send the command back to the **Controller** with the resetState=true flag	As of 0.4.1.1, The destinations for the State updates are defined when editing the State the Bolt Editor windows. There are options for the replication destination, such as "replicate to everyone" (most useful typically with non-auth case), "Don't replicate to controller" (useful in authoritative case) etc. Again, remember that State changes can only be made from the Owner. There is a hope of a special API to make this a little more convenient..	**Global**: These are sent to all nodes - classes that derive from `Bolt.GlobalEventListener()` **Entity**: These are sent to the nodes for this entity - classes that derive from `Bolt.EntityBehaviour<IYourState>`
Reliability See http://doc.photonengine.com/en/bolt/current/in-depth/reliability	Commands are sent in a best-effort manner, subject to network availability. They are completely unreliable, however they are sent with some redundancy, meaning that each command will be sent in several packets to give it a higher chance of arriving. If an entity **Owner** has not received a Command from a **Controller** they can use the `MissingCommand()` callback to detect this and potentially create a default action. Command ordering however is reliable: There may be lost Commands but the ones that arrive will always be delivered in the order they were queued with the `QueueInput()` call.	For STATE changes, ordering is unreliable, but eventual delivery of all changes is reliable. Put another way, State updates are not reliable in the way that everything comes in the exact order you change it (aka TCP style reliability), but they are reliable in the way that all changes will eventually make it across, even if they are lost on the first attempt.	**Global** - global events can be either sent "reliably" (default: `Bolt.ReliabilityModes.ReliableOrdered`) or "unreliably" (`Bolt.ReliabilityModes.Unreliable`) **To an Entity** - entity events are always sent "unreliably" (src) For EVENTS, the definition of 'reliable' behavior is as follows: Unreliable events means "It may not get there, but everything that gets there will be in the order it's sent (with any missed events/calls obviously not getting there) Reliable Events in "bolts" terms means = It will always get there, in the exact order you sent it.

Code example for send:	`public override void SimulateController()`
	`{`
	`  PollKeys();`
		State changes will ONLY be replicated if made on the node that is the **Owner** of the Entity.	**Global**: (see as image) (src)
			`using (var evnt = ChangeDoorState.Raise(Bolt.GlobalT`

<table>
<tr>
<td></td>
<td>

```
  IPlayerCommandInput input;
  input = PlayerCommand.Create();
  input.forward = forward;
  input.backward = backward;
  input.left = left;
  input.right = right;
  input.jump = jump;
  input.Token = new TestToken();
  entity.QueueInput(input);
}
```

</td>
<td>

The State changes are typically made in ExecuteCommand() when invoked on the entity's Owner.

- Changes to Transforms and Animations should be made directly on the entity.transform and entity.animation.
- Changes to other properties are done using code such as

```
using(var mod =
state.Modify())
{
    mod.Speed = curSpeed;
}
```

state.Modify() has been removed along with using (var evnt)

</td>
<td>

```
 argets.Everyone))
 {
   evnt.isOpen = b;
 }

 // Example of Global unreliable
 using (var ev =
 LogEvent.Raise(Bolt.ReliabilityMod
 es.Unreliable))
```

To Entity: (see as image) (src)

```
var casingEvnt =
SpawnCasing.Raise(entity);
    casingEvnt.cPosition =
weaponStatsScript.caseEjectionPoint
.position;
    casingEvnt.gunRef =
correctIndex;
casingEvnt.Send();
using (var evnt) has been removed
```

</td>
</tr>
<tr>
<td>Code example for receive:</td>
<td>

```
public override void
ExecuteCommand(Bolt.Command c, bool
resetState)
{
  PlayerCommand cmd;
  cmd = (PlayerCommand) c;
  if (resetState)
  {
     // This can happen on Controller
    _motor.SetState(
       cmd.Result.position,
       cmd.Result.velocity,
       cmd.Result.isGrounded,
       cmd.Result.jumpFrames);
  }
  else
  {
   // move & save resulting state
   var result =
_motor.Move(cmd.Input.forward,
cmd.Input.backward, cmd.Input.left,
cmd.Input.right, cmd.Input.jump,
cmd.Input.yaw);

  // etc...
  }
  ...
}
```

There is also a flag `command.isFirstExecution()` which is important for trigger-type situations such as 'pushed jump', 'fired shot' etc. In general, Bolt rewinds and redoes Commands multiple times in order to support authoritative movement even in the face of network lag/errors. The `isFirstExecution` flag can be used in `ExecuteCommand()` to isolate code that you want to ensure will only be invoked exactly once for each Command. (TODO - need to confirm that means once on Controller, and once on Owner)

</td>
<td>

Bolt will automatically apply States that have been 'attached' (aka 'linked') to a BoltEntity - this is how Animations and Transforms must be applied. if they have not been attached, then they will have no effect.

You must to attach (link) your Transform, Animator etc when your State is attached, eg (in a script that inherits from `Bolt.EntityBehaviour<IMyState>`):

```
public override void Attached ()
{

state.transform.SetTransforms(trans
form);

state.SetAnimator(GetComponentInChi
ldren<Animator>()); // Or wherever
your animator is
}
```

After that, if your State properties are all set up properly in-editor, then all you have to do is change the linked properties, and your Transform, Animator (etc) should replicate as intended.
(In-editor example of Mecanim properties in state)

Then, to change other properties of the State: In a script that inherits from: `Bolt.EntityBehaviour<IMyState>`

```
state.Speed = curspeed;
```

Then, as long as the controller is the only one running that bit of code (eg: wrap that in a `if (BoltNetwork.isOwner)` clause), Bolt will replicate that variable across the network, and push it into that object/animator/etc automatically, if your State properties are set up as shown above. (src, src)

It is also possible to have callbacks:

```
state.AddCallback("Speed",
OnSpeedChanged)
state.RemoveCallback("Speed",
OnSpeedChanged)

void OnSpeedChanged()
{
    // do stuff
}
```

(See additional notes in next section)

```
??? What about  - TODO
IState.SetDynamic(prop, value)
```

</td>
<td>

Global: Make sure your script inherits from `Bolt.GlobalEventListener`

Entity: Make sure your script inherits from `Bolt.EntityEventListener` / `Bolt.EntityEventListener<StateNameGoesHere>` (These inherit from `Bolt.EntityBehaviour` already)

Then, to receive an event:

```
public override void OnEvent
      (ChangeDoorState evnt)
{
   if (evnt.isOpen == true)

      SendMessage("PlayLinkedSound"
      );
}
```

To access State from a global event, pass the state in the event and use something like `evnt.entity.GetState<TState>().Property;`
where `TState` is your state Type, for example `IPlayerState`

</td>
</tr>
<tr>
<td>More info</td>
<td></td>
<td>http://doc.photonengine.com/en/bolt/current/reference/state-replication</td>
<td></td>
</tr>
</table>

TODO: Add notes on read&pack

State callbacks

One can define callbacks on arbitrary types in the State object:

```
state.AddCallback("myStateProperty", myCallbackFunction)
```

Note that triggers are a C# Delegate and don't require the state.AddCallback() method to be used. Instead you use the default adding to a delegate. Like so:

```
state.myTrigger += myCallbackFunction
```

State Callbacks on arrays

You can set up a callback on an array in an array also, for example

```
state.AddCallback("InventorySlots[]", myCallbackFunction)
```

The callback will be invoked once for each property/element that changed. You can know the element and property that changed if you use the extended callback which is

```
PropertyCallback(IState state, string propertyName, ArrayIndices indices);
```

or you can hook it to just a sub property by using "InventorySlots[].Quantity" as the first parameter of `AddCallback()`.

The 'indices' parameter is plural because you can have nested arrays e.g. `InventorySlots[].Enchants[] ...`
Say that you hook a callback to `InventorySlots[].Enchants[].TypeOfEnchant`. When that changes, array indices will contain the index into `InventorySlots[]` and then into `Enchants[]` so you know which property at what indices changed. (example here)

There is no way to collapse down these callbacks into one invocation.

Good opportunities to do the `AddCallback()` are:

- If you want the callback to fire on all nodes, then the `Attached()` callback on the entity would be a good place.
- If you want the callback to only fire on the **Controller**, then the `ControlGained()` callback on the entity would be a good place. It can't go in `Attached()` since we don't know at `Attached()` time if the node is going to be a **Controller**.

See http://doc.photonengine.com/en/bolt/current/reference/state-callbacks for fuller explanations of State callbacks.

An additional but important 'message type': IProtocolToken during BoltNetwork.Instantiate()/Attached() and Connect()

See also http://doc.photonengine.com/en/bolt/current/reference/iprotocoltoken

`IProtocolToken` during Instantiation/`Attached()`:

When Entities are instantiated, information can be passed from the **Owner** (who is making the instantiation request) to the other nodes. They will receive the information in their `Attached()` callback. `IProtocolToken` is useful in `Attached()` since it has guaranteed delivery, whereas Commands do not.

An example of a simple Token is as follows

```
public class MyTokenEntity : IProtocolToken
    {
        public ushort  NetworkEntityId { get; private set; }

        public virtual void Read (UdpKit.UdpPacket packet)
        {
            NetworkEntityId = packet.ReadUShort ();
        }

        public virtual void Write (UdpKit.UdpPacket packet)
        {
            packet.WriteUShort (NetworkEntityId);
        }

        public MyTokenEntity ()
        {
        }

        public MyTokenEntity (ushort networkEntityId)
        {
            NetworkEntityId = networkEntityId;
        }

        public override string ToString()
        {
            return "NetworkEntityId: " + NetworkEntityId;
        }
    }
```

The Token has to be registered with Bolt before it can be used - for example in the `BoltStarted()` global callback..
`BoltNetwork.RegisterTokenClass <MyTokenEntity>()` on both the clients and server. An example of this is in the tutorial at bolt_tutorial/scripts/Callbacks/TokenCallbacks.cs

When your code requests a Bolt Entity be instantiated using `BoltNetwork.Instantiate()` on the (now, implicitly) **Owner** of the entity, it can send in the Token that it creates. On the other nodes, override `Attached()` with the version that takes an `IProtocolToken` and cast it to a `MyTokenEntity`

`IProtocolToken` during Connection request/allow

See http://doc.photonengine.com/en/bolt/current/in-depth/protocol-tokens, but the key steps are:

1. Set Bolt to require manual acceptance of connections: Use Window > Bolt engine > Settings. Then change "Accept Mode" to Manual.

2. Define a user Token class that has the information you need

3. At some point in your client connect code, have something like:

```
BoltLauncher.StartClient();
UserToken userToken = new UserToken ( usernameString );
BoltNetwork.Connect ( UdpEndPoint.Parse (serverIPAddressString), userToken );
```

4. Some global callback code to (a) register tokens, and (b) accept and remember the connections:

```
public class LocalUserToken {
  public string username;
  public string connectionTime;

  public LocalUserToken ( UserToken userToken, string connectionTime ) {
    this.username = userToken.username;
    this.connectionTime = connectionTime;
  }
}

[BoltGlobalBehaviour]
public class TokenCallbacks : Bolt.GlobalEventListener
{
  public override void BoltStarted ()
  {
    // Note that this will be called on Client and Server
    BoltNetwork.RegisterTokenClass<UserToken>();
  }

  public override void ConnectRequest( UdpKit.UdpEndPoint endPoint, IProtocolToken token )
  {
    // Note that this is a Server-only callback, so will not be invoked on Client nodes.
    UserToken userToken = (UserToken)token;

    if ( userToken.username.Equals("Dave") ) {
      // Don't let Dave back on the server
      BoltNetwork.Refuse( endPoint );
    } else {
      // Accept the connection and add a userToken to the connection
      BoltNetwork.Accept( endPoint, new LocalUserToken(userToken, System.DateTime.UtcNow.ToString()) );
    }
  }
}
```

How Commands become State changes

There are exactly two mechanisms that replicate changes to an entity's State:

a. Your `ExecuteCommand()` callback can have some code that updates State
   i. Note that the syntax has changed to use getters/setters as of Bolt 4.1.x.x. In the Bolt Tutorial this happens in the `motor.Move()` methods which is invoked by `ExecuteCommand()`.
   ii. Bolt will only replicate State changes for an Entity when they change on the node that is the **Owner** for this entity.

b. For transforms and animations (only) there is a special binding of State to the BoltEntity's parameters that you should typically set up in the your `Attached()` callback - using `state.SetTransform()` and `state.SetAnimator()`
   i. For this reason, there is no way to update fields of type Transform or Animation on State objects.

TODO - describe the Command and State's 'Replication' parameter and destinations choices (e.g. 'Everyone Including Controller' etc) -- update and doc the 0.4.1.x behaviors.

3. Bolt Code and Objects: Classes, Singletons, Callbacks, GameObjects, Components and Scripts

In this section, we describe how the Bolt code is connected. We start from the global statics/singletons for working with the system, Callbacks for plugging into key state changes, and then how Bolt uses/extends Unity GameObjects and Components/Scripts.

Common Bolt Super-Classes that your code will sub-class

In general, one should use Composition more than Inheritance, but there are areas where inheritance is the core way to connect into the Bolt system. These common superclasses are as follows.

Super-Class	**Purpose and Usage**
`Bolt.EntityBehaviour<IYourState>`	The Base class for Unity behaviours
`Bolt.EntityEventListener<IYourState>`	The Base class for Unity classes that want to access/hook Bolt methods for dealing with single Entities - this is where most of your per-entity (npc, player, items etc) code will go in order to keep it modular. Callbacks available when inheriting from `Bolt.EntityEventListener<IYourState>` include:

<table>
<tr>
<td></td>
<td>

```
/// Invoked when the entity has been initialized, before Attached
/// This is when we should link Bolt State to Game Object
public virtual void Initialized() { }

/// Invoked when Bolt is aware of this entity and all internal state
/// has been setup. Called on Owner, Controller and Proxy
public virtual void Attached() { }

/// Invoked when this entity is removed from Bolts awareness
public virtual void Detached() { }

/// Invoked each simulation step on the owner
public virtual void SimulateOwner() { }

/// Invoked each simulation step on the controller
// On Controller, this is when we create a Command to send to Owner
public virtual void SimulateController() { }

/// Invoked when you gain control of this entity
public virtual void ControlGained() { }

/// Invoked when you lost control of this entity
public virtual void ControlLost() { }

/// Invoked on the owner when a remote connection is controlling this
//// entity but we have not received any command for the current
//// simulation frame.
public virtual void MissingCommand(Bolt.Command previous) { }

/// Invoked on both the owner and controller to execute a command
/// On Controller and Owner, apply the Command. Note that the Controller
/// has more complex sub-cases to handle (resetState, firstCall etc)
public virtual void ExecuteCommand(Bolt.Command command, bool resetState) { }
```

These are described in more detail in the ‘callbacks’ section below

</td>
</tr>
<tr>
<td><code>Bolt.GlobalEventListener</code></td>
<td>

This is what you inherit from in order to get callbacks on GLOBAL (rather than per-entity) situations. A good example is the SceneLoad-related callbacks. Note that it is also to get some callbacks that are about entities.. ie. ControlOfEntityLost(entity) is the global equivalent of the per-entity ControlLost() callback. This allows you to put more common behaviours into a common place in your code.

Note that this class has [...] attributes that are specified on the Callback functions to determine the contexts in which it will be invoked: Server/Client, Specific Unity scenes, etc

Callbacks available when inheriting from <code>Bolt.GlobalEventListener</code> include:

```
  public override void BoltStartDone() and BoltStartBegin()
  public override void BoltShutdown() REPLACED in 0.4.3 with
BoltShutdownBegin(Bolt.AddCallback registerDoneCallback)
  public override void PortMappingChanged()
  public override void ControlOfEntityGained(BoltEntity a)
  public override void ControlOfEntityLost(BoltEntity a)
  public override void EntityAttached(BoltEntity entity)
  public override void EntityDetached(BoltEntity entity)
  public override void EntityReceived(BoltEntity entity)
  public override void OnEvent(myEvent evnt)
  public override void SceneLoadLocalDone(string map)
  public override void ConnectAttempt(endpoint)
  public override void ConnectFailed(endpoint)
  public override void ConnectRefused(endpoint)
  public override void Disconnected(connection)

  public override void SceneLoadLocalBegin(string map)
  public override void SceneLoadRemoteDone(BoltConnection c)
  public override void SceneLoadLocalDone(string map)

  public override void ZeusConnected(UdpKit.UdpEndPoint endpoint)
  public override void ZeusConnectFailed(UdpKit.UdpEndPoint endpoint)
  public override void ZeusDisconnected(UdpKit.UdpEndPoint endpoint)
  public override void ZeusNatProbeResult(UdpKit.NatFeatures features)
  public override void SessionListUpdated(Map<System.Guid, UdpSession>sessionList)
  public override void SessionConnectFailed(UdpSession session)

  // Server-only (???):
  public override void ConnectRequest(endpoint, token)
  public override void Connected(connection, token)
  public override void Disconnected(connection)
```

These are described in more detail in the following section

</td>
</tr>
<tr>
<td><code>Bolt.BoltSingletonPrefab<PlayerCamera></code></td>
<td>TODO</td>
</tr>
<tr>
<td><code>Bolt.IProtocolToken</code></td>
<td>See the separate section on <code>IProtocolToken</code> in this doc. This is used to pack/unpack some data structures that aren’t the Bolt standard ones. A common example is login/password</td>
</tr>
</table>

It is possible to add multiple components and sub-objects to a Bolt Entity Game Object, for example then finding these using `GetComponent<IYourDesiredInterfaceOrTypeHere>()`. Take that approach for adding behaviors when possible.

Bolt Callbacks

Global and Entity contexts for callbacks

Callbacks in this context means the methods in your class that Bolt will invoke to involve your code in Bolt's network logic/communication processing. There are two main categories of these callbacks - Global ones (e.g. scene-related etc) and entity ones (e.g. npc or item related):

Callback Type:	**Bolt Global**	**Bolt Entity**
Purpose of callback type	Intended for activities that are related to the overall game.	Intended for activities that are related specifically to an entity
Example use case	Chat, Explosion, Cinematics	Fire, Health changing, Jumping.
How to define	In Unity Editor "Window / Bolt / Assets " - then define Commands, States, Events, Structs to represent what will be communicated over the network (and relative priority)	
How to access	Your class must inherit from `Bolt.GlobalEventListener` Then, the Bolt callbacks are typically defined just as a `public void override callbackMethodName()` There are two ways to get Bolt to detect your callback script and invoke the methods on it, the most basic one is to just do the default 'Unity thing' and attach it to a game object somewhere in a scene like you would with any script. The other way is to use the [BoltGlobalBehaviour] attribute, if you specify this attribute on your callback class, like below, Bolt will automatically find your class and create an instance of it that lives with together with Bolt and is destroyed when Bolt is shutdown. States & Commands: Not applicable, since they are entity concepts Events: You have to inherit your class from `Bolt.GlobalEventListener` and also override the `OnEvent()` function.	Your class must inherit from `Bolt.EntityEventListener<IYourBoltStateForThisEntity>` Then, the Bolt callbacks are typically defined just as a `public void override callbackMethodName()` Commands: See discussion above about `SimulateController()` and `ExecuteCommand()` States: See discussion above about `ExecuteCommand()` and `Attach()`. Events: You have to inherit your class from `Bolt.EntityEventListener<IYourBoltStateForThisEntity>` and override the `OnEvent()` function.

Example of receiving these callbacks	`[BoltGlobalBehaviour(BoltNetworkModes.Server, "L01")]`
	`public class PlayerCallbacks : Bolt.GlobalEventListener`
	`{`
	`  GameObject _playerCam;`
	`  public override void SceneLoadLocalDone(string map)`
	`  {`
	`  // etc...`
	`  }`
	`}`
	
	Note that the `[..] Attribute` of `BoltGlobalBehaviour` limits the context of when this callback will be invoked. In this case Server, and only on the level called "L01".
		TODO
		
		Events: register a callback via `AddCallback("state", stateChanged)`
		
		If using triggers, register the callback using `state.yourTrigger += yourTriggerFunction;`

More Info	See also http://doc.photonengine.com/en/bolt/current/in-depth/global-callbacks	

Execution order and responsibilities of Bolt Callbacks

For diagrams,

- See the Unity docs on Unity callback execution order which includes a nice flowchart of the Unity-only callbacks.
- See the Bolt Update loop flowchart

Here is that information in more detail as a table - it lists the main Unity and Bolt callbacks and their sequencing and responsibilities:

Callback (in execution order)	**Caller**	**Category**	**Notes**
`BoltStarted()`	**Bolt (Global)**	global-init	This function is called on *Client* and *Server* (subject to the usual annotation filters for Global Events). Things to do in this callback include: ● Registering Token classes - e.g. `BoltNetwork.RegisterTokenClass<MyTokenEntity>()` (example)
`Awake()`	Unity	entity-init	This function is always called before any Start functions and also just after a prefab is instantiated. (If a GameObject is inactive during start up `Awake()` is not called until it is made active, or a function in any script attached to it is called.) Things to do in this callback include: ● Caching component reference lookups and other static data
`OnEnable()`	Unity	entity-init	(only called if the Game Object is active): This function is called just after the object is enabled. This happens when a MonoBehaviour instance is created, such as when a level is loaded or a GameObject with the script component is instantiated. Things to do in this callback include: ● (Usual Unity stuff)

`Initialized()`	**Bolt (Entity)** **(all)**	entity-init	This happens before `Attached()`. It is not commonly required, but is in Bolt to allow you to do sort of pre-attached initialization when Bolt is first aware of the object but it's still not attached to the nodes and wired up. It can be useful when adding Bolt to games built without network support, but not essential.
`Attached()`	**Bolt (Entity)** **(all)**	entity-init	This is invoked when Bolt is aware of this entity and all internal state has been setup. `Attached()` is called on the entity **Owner** before the entity is sent over the network (so you can set initial state) and it is called on the **non-Owner** nodes when the new network entity is instantiated on those nodes (so you can sync the initial state back to your business layer) Things to do in this callback include: ● Process any entity information sent in an `IProtocolToken` by the **Owner** via `BoltNetwork.Instantiate()`. `IProtocolToken` is useful on `Attach()` for this kind of information. See the other section in this doc on `IProtocolToken` for more info. This is a good way to passing names for example. ● Bind any transforms to the State using `state.SetTransform()` ● Bind any animations to the State using `state.SetAnimator()` ● Set any animation weights using `state.Animator.SetlayerWeight()` ● Set any callbacks on state changes, e.g. `state.OnFire += OnFire` or `state.AddCallback("weapon", WeaponChanged)` ● Set/Apply any other initial state (especially important on the **Owner**)
`EntityAttached()`	**Bolt (Global)** **(all)**	entity-init	This is the Global counterpart of the entity.`Attached()` callback. It is invoked on the node that is being Attached, so can be for any Entity role. Note that the entity role has not been fully determined at this time, so `entity.hasControl` should not be used/checked at this time (though `entity.isOwner` should be valid since that was decided before `Attached()`. Code that you want to initialize for **Controllers** is best placed in `ControlGained()`.
`Start()`	Unity	entity-init	`Start()` is called before the first frame update only if the script instance is enabled. Things to do in this callback include: ● (Usual Unity stuff) `Start()` is not a good place to do initializations for Bolt. Use `Awake()` instead
`SceneLoadLocalBegin()`	**Bolt (Global)**	scene-load	This callback is called when a local scene load begins. This callback can be invoked on *Client* or *Server* - see the 'extra notes on scene loading' section below for more context. Things to do in this callback include: ● Show a Loading screen/popup]
`SceneLoadLocalDone()`	**Bolt (Global)**	scene-load	This callback is called once a local scene load is complete. This callback can be invoked on *Client* or *Server* - see the 'extra notes on scene loading' section below for more context. Things to do in this callback include: ● Stop showing a loading screen/popup ● Instantiate Game UI and Camera for the player (as in PlayerCallbacks.cs) ● Instantiate any entities for this scene that this node wants to be **Owner** for. It is very important that you do not attach this behavior to any GameObject in any of your scenes.
`SceneLoadRemoteDone()`	**Bolt (Global)**	scene-load	This callback is called once a remote scene load is complete. A *Client* will only receive this once - saying that the *Server* has completed loading the Scene. A *Server* will get this once for each *Client* connection. Since `BoltNetwork.LoadScene(mapName)` can only be invoked on the *Server*, then this `SceneLoadRemoteDone()` will be called on the the remote (client) node at the end on the connection that is being passed in has loaded the scene. Things to do in this callback include: ● Instantiate any per-connection entities for this scene that this node wants to be **Owner** for. The most common case is the Player entity, in the case of Authoritative *Server* for the player. It is very important that you do not attach this behavior to any GameObject in any of your scenes.
`ControlGained()`	**Bolt (Entity)** **Controller**	initializations	This callback is called on the **Controller** node once **Control** has been assigned for this entity by the **Owner** using `TakeControl()` or `AssignControl()`. It is invoked on the node that now has Control for this entity. Things to do in this callback include: ● Add state callbacks to do stuff for the player based on state changes (See example in PlayerSfx.cs)
`ControlOfEntityGained (BoltEntity a)`	**Bolt (Global)**	initializations	This is the Global counterpart of the entity.`ControlGained()` callback. It is invoked on the node that now is the **Control** for this entity. Things to do in this callback include: ● For player.. add audio listener, set camera callbacks/target (as done in PlayerCallbacks.cs for example) Note that the Bolt tutorial as of 0.4.1.5 has some code in `PlayerCallbacks.cs` using this callback that would probably be cleaner to put in `ControlGained()` in `PlayerController.cs`

`ControlLost()`	**Bolt (Entity)** **Controller**	teardowns	This callback is called once **Control** has been revoked for this entity by the **Owner**. It is invoked on the node that is losing **Control** for this entity. Things to do in this callback include: • Remove state callbacks to do stuff for human player based on state changes (See example in PlayerSfx.cs)
`ControlOfEntityLost (BoltEntity a)`	**Bolt (Global)**	teardowns	This is the Global counterpart of the entity.`ControlLost()` callback. It is invoked on the node that is losing **Control** for this entity.
`Update()`	Unity	game loop	The `Update()` callback is called once per frame. It is the main workhorse function for frame updates. Note that Input typically updates in `Update()` - see the tutorial PlayerController.cs for an example of recording input state in the Object, then applying it in Bolt's `SimulateController()` callback via a Command. Things to do in this callback include: • For a Player-type entity, get input, and store the input intent from the user locally in this script
`FixedUpdate()`	Unity	game loop	`FixedUpdate()` is often called more frequently than `Update()`. It can be called multiple times per frame, if the frame rate is low and it may not be called between frames at all if the frame rate is high. All physics calculations and updates occur immediately after FixedUpdate. When applying movement calculations inside FixedUpdate, you do not need to multiply your values by Time.deltaTime. This is because FixedUpdate is called on a reliable timer, independent of the frame rate.The `FixedUpdate()` callback does **not** happen every frame. Note that Bolt runs (and hence invokes many of it's callbacks) during Unity's `FixedUpdate()` - this has different frequency of calling to `Update()` so your code should **not** assume 1:1 invocation of `Update()` then `SimulateController()` for example. Things to do in this callback include: • Entity: Directly update transforms/animations for server-time-positioned objects (e.g. Elevator.cs in the bolt tutorial) • Global: Invoke a player spawn (e.g ServerCallbacks.cs)
`SimulateOwner()`	**Bolt (Entity)** **Owner**	game loop	Called by Bolt if this node is the **Owner** for this entity. Things to do in this callback include: • This is a good place to directly change State on the **Owner**. (e.g. PlayerController.cs uses this to do slow-heal of the Player) Note: `SimulateOwner()` and `SimulateController()` execute on a fixed frequency defined in Window/Bolt Engine/Settings.
`SimulateController()`	**Bolt (Entity)** **Controller**	game loop	Called by Bolt if this node is the **Controller** for this entity. Things to do in this callback include: • For a Player-type entity, use the player input that was stashed in `Update()` to build a Command that will then be queued for `ExecuteCommand()`. Note: `SimulateOwner()` and `SimulateController()` execute on a fixed frequency defined in Window/Bolt Engine/Settings.
`MissingCommand()`	**Bolt (Entity)** **Owner**	game loop	This is called by Bolt on the entity **Owner**. The purpose is to let the Owner inject replacement Commands in case a Command hasn't been received in time by the **Owner** from a **Controller**. In this callback, the **Owner** would supply a "null command" and queue up in the same way `SimulateController()` does. Note that as of Bolt 0.4.1.5 the Bolt tutorial does not use this (but it should).
`ExecuteCommand()`	**Bolt (Entity)** **Owner + Controller**	game loop	Called by Bolt if this node is the **Owner** or **Controller** for this entity. Things to do in this callback include: • invoke a Motor to update state based on the requested input (or result if `resetState==true`). • Store the resultant state in `Command.result` (if `resetState==false` and `entity.isOwner==false`). Note that proxies never get called with `ExecuteCommand()` so if you want to show effects from entities such as explosions/damage/shots, then you need another way to inform the proxy that this has occurred (i.e. an *Entity event* or a callback on a state parameter changing.
`LateUpdate()`	Unity	game loop	`LateUpdate()` is called once per frame, after `Update()` has finished. Any calculations that are performed in `Update()` will have completed when `LateUpdate()` begins. Things to do in this callback include: • A common use for `LateUpdate()` would be a following third-person camera. If you make your character move and turn inside `Update()`, you can perform all camera movement and rotation calculations in `LateUpdate()`. This will

			ensure that the character has moved completely before the camera tracks its position.
`OnEvent()`	**Bolt (Global)** **Bolt (Entity)**	game loop	This is used for receiving custom events that you define using Bolt 'Event' assets, and that you send using `BoltNetwork.Raise()` for global events, or `entity.Raise()` for entity events. TODO: Confirm where this fits in sequence of Bolt calls. Entity unreliable, global always reliable
`OnDestroy()`	Unity	teardowns	This function is called after all frame updates for the last frame of the object's existence (the object might be destroyed in response to Object.Destroy or at the closure of a scene).
`Detached()`	**Bolt (Entity)** **All**	teardowns	Invoked when this entity is removed from Bolts awareness. It is invoked on all entity roles (**Controller**, **Owner**, **Proxy**). Things to do in this callback include: • Updating minimap/hud etc that some entity is no longer around • Disconnect message about other player
`EntityDetached()`	**Bolt (Global)**	teardowns	This is the Global counterpart of the entity.`Detached()` callback. It is invoked on all entity roles (**Controller**, **Owner**, **Proxy**).
`PortMappingChanged()`	**Bolt (Global)**	?	TODO. Seems related to NAT
`BoltShutdown()`	**Bolt (Global)**	teardowns	This function is called when/after Bolt is shutdown.

Note that the above table did not include the `ConnectAttempt()/ConnectFailed()/ConnectRefused()/Disconnected()` and `ConnectRequest(endpoint, token)/Connected(connection, token)/Disconnected(connection)` callbacks.
TODO - Connection callbacks

Extra notes on Scene Loading callbacks: `SceneLoadLocalDone()` and `SceneLoadRemoteDone()`

Only a *Server* can call `BoltNetwork.LoadScene()`. The sequence of callbacks after this is as follows:

- The Server will receive the `SceneLoadLocalDone()` callback when it is locally (i.e. on the Server) done,
- Each Client will receive the `SceneLoadLocalDone()` callback when the scene has been loaded on that client node
- The Server will receive the `SceneLoadRemoteDone(connection)` callback when the scene has been loaded on each client node
- Each Client will then receive the `SceneLoadRemoteDone(connection)` callback after the *Server* has processed it's `SceneLoadLocalDone()`

In addition, for a given node, it is guaranteed to execute in the order of: 1) `SceneLoadLocalDone` and then 2) `SceneLoadRemoteDone(connection)` EVEN if the remote finishes before you are locally done (because knowing that the remote is done without knowing that you yourself is done, is mostly useless).

A Client will only ever see one `SceneLoadRemoteDone(connection)` callback for a given `LoadScene()` cycle - i.e. the *server's* one. It doesn't see this for all the other *client* nodes

Note that in the ServerCallbacks.cs script in the Bolt tutorial, this code is guarded by a decorator that limits the invocation to only Server side:

```
[BoltGlobalBehaviour(BoltNetworkModes.Server, "Level1")]
```

..so that the Player characters are only Instantiated from the *Server* - hence the Players are all **Owned** by the the *Server* and so Players are 'authoritative *server'* entities. The `SceneLoadRemoteDone()` callback allows the *Server* to instantiate a Player per connection, once that remote scene has been loaded.

Bolt Singletons & statics - always there when you need them

Class to use	General purposes	Cool things you can use or call with it

`BoltNetwork`	• Instantiate/Destroy Bolt Entities
	• Get local/remote frame stats
	• Accept/reject connections (Server)
		`BoltNetwork.Connect()`
		`BoltNetwork.Accept(endpoint)`
		`BoltNetwork.RegisterTokenClass<type>()`
		
		`BoltNetwork.Instantiate(BoltPrefabs.YourPrefabsID, …)`
		`or BoltNetwork.Instantiate(prefabGameObject, …)`
		
		`BoltNetwork.Destroy(entity.gameObject)`
		
		`BoltNetwork.isClient`
		`BoltNetwork.isServer`
		
		`BoltNetwork.serverTime`
		`BoltNetwork.serverFrame`
		`BoltNetwork.frame`
		`BoltNetwork.frameDeltaTime`
		`BoltNetwork.framesPerSecond`
		
		`BoltNetwork.LoadScene()`
		
		`BoltNetwork.scopeMode`
		
		`using (var hits = BoltNetwork.RaycastAll(new Ray(pos, look * Vector3.forward), cmd.ServerFrame)) { … }`

`BoltNetworkInternal`	• `BoltLauncher.StartServer()` uses these to initialize a Server	You probably want to leave `BoltNetworkInternal` alone until you really need a custom launcher

`PlayerCamera`	• Create a Player Camera	`PlayerCamera.Instantiate()` The camera Prefab Asset must be named “PlayerCamera” and be placed in a folder named "Resources" so it can be found using Unity’s `Resources.Load()`
`BoltPrefabs`	• A convenient way to reference (for Invocation) a Bolt Entity Prefab by name without it being in a Resources/ folder.	`BoltNetwork.Instantiate(BoltPrefabs.YourPrefabsID, ...)`

`BoltLauncher`	• Startup (*Client* and *Server*)
	• Disconnect (*Client* and *Server*)
		`BoltLauncher.StartServer();`
		`BoltLauncher.StartClient();`
		`BoltLauncher.Shutdown();`

TODO more…?		

Bolt GameObjects - Prefabs, Parenting and more

Making Bolt Entities as Unity Prefabs

For GameObjects to have replicated state (and other goodness) in Bolt, there must be a BoltEntity component attached to the Game Object. Also, it is almost always the case that these Bolt Entity Game Objects are Unity Prefabs. *fholm* has said that it’s *possible* to use Bolt entity without prefabs (and for example he has done that in the game he worked on), but it’s not normal (and he implied that ‘here be dragons’). So unless you are a BoltEngine expert, keep life simple by assuming BoltEntity is always used on Unity Prefabs, and instantiated using BoltNetwork.Instantiate().

When defining these Bolt Entity Prefabs:

1. Make your Unity prefab, view it in the inspector
2. Add the Bolt Entity Component - typically to the root Game Object in your prefab
3. Use *Unity Menu -> Assets -> Bolt Engine -> Rebuild Prefab Database* to correct the Prefab ID
4. Assign a serializer to the Bolt Entity Component (this will define which parameters Bolt will replicate for this Bolt Entity):
    a. Define a new Bolt State using Unity WIndows -> Bolt Engine -> Bolt Assets -> (right click) -> new State
    b. Save the State
    c. Use Unity Menu -> Assets -> Bolt Engine -> Compile Assembly
    d. Now (finally) you can assign the serializer in the Bolt Entity component of your Prefab
5. Make a note to remember to write an Attach() method as described above in the State section. Look for the section ‘Code example for receive’ which gives examples of the Attach() methods.

Parenting, deparenting and reparenting BoltEntities (TODO)

Unity has had a long-awaited mystical feature called Nested prefabs, which still hasn’t shipped, many years after first being mentioned. However, Bolt does have a way to dynamically reparent Entities using `entity.SetParent(anotherBoltEntity)`

- **Parenting of Bolt Entities:**
    - **fholm says that reparenting still has some issues as of 0.4.1.x. Need to investigate what works, what doesn’t.**
- **Reparenting of Bolt Entities**:
    - This can be done via `yourBoltEntity.SetParent(anotherBoltEntity)`
    - Only the owner or a controller *with local prediction* can reparent an Entity
- **De-parenting of Bolt Entities**:
    - There was an issue in 0.4.0.19 that prevented deparenting. (TODO - check this is fixed)

Don’t make BoltEntity’s GameObject inactive

Don't set GameObjects with a Bolt Entity on them, or on a child of them, to inactive (make sure the BoltEntity is on a parent GameObject that doesn't get set to inactive).

If you disable the GameObject the BoltEntity script is on, then Bolt will destroy the BoltEntity.

if you want to inhibit certain functionality for a GameObject/Entity, options include:

- To temporarily stop replication traffic in Bolt, use `entity.Freeze()` (See http://doc.photonengine.com/en/bolt/current/in-depth/freeze-idle-setscope)
- To disable physics (for example for rigidbodies that are not currently interesting), set the GameObject to be InverseKinematic (Example in Recipe 001)
- To disable everything , the best approach is to have the GameObject you want to disable be a child of the GameObject with the BoltEntity.

Setting the UniqueId for a Bolt Entity

The **Owner** of the Bolt Entity is the only one that can set the Unique ID.
The Unique ID has to be manually set with yourBoltEntity.SetUniqueId(Bolt.UniqueId.New());
// TODO - Validate that this is still the case in 0.4.0.22 and 0.4.1.x

Typical naming conventions and responsibilities for your Bolt-related scripts

These are the conventions you see in the Bolt Tutorial, and are used in most descriptions within the Bolt community. They approximately map to wider Unity and Game industry use with networked game systems, but not exactly..!

Area	**Naming convention**	**Typical Responsibilities**
Callbacks	server_callbacks.cs	TODO - scene loads etc

<table>
<tr><td>Callbacks</td><td>player_callbacks.cs</td><td>This is where Game UI, Cameras, Audio Listeners etc are instantiated. For example:

<pre>
[BoltGlobalBehaviour("Level1")]
public class PlayerCallbacks : Bolt.GlobalEventListener {
  public override void SceneLoadLocalDone(string map) {
    // ui
    GameUI.Instantiate();

    // camera
    PlayerCamera.Instantiate();
  }

  public override void ControlOfEntityGained(BoltEntity arg, Bolt.IProtocolToken token)
{
    BoltLog.Info("ControlGained-Token: {0}", token);

    // add audio listener to our character
    arg.gameObject.AddComponent<AudioListener>();

    // set camera callbacks
    PlayerCamera.instance.getAiming = () => arg.GetState<IPlayerState>().Aiming;
    PlayerCamera.instance.getHealth = () => arg.GetState<IPlayerState>().health;
    PlayerCamera.instance.getPitch = () => arg.GetState<IPlayerState>().pitch;

    // set camera target
    PlayerCamera.instance.SetTarget(arg);
  }
}
</pre></td></tr>
<tr><td>Bolt Entity</td><td>yourEntityName_Controller.cs</td><td>*** ALERT!! BEWARE OF CONFUSING NAMING COLLISION HERE ***
This does <u>not</u> mean Controller in the ‘bolt’ sense. It means controller in the wider sense of ‘code that controls the Entities behavior’. Some of this code will be executed on a Bolt Controller / Owner / Proxy - see the definitions and tables above to understand exactly which ones.

This typically subclasses Bolt.EntityEventListener<IPlayerState> which in turn subclasses BoltEntity. See the notes on BoltEntity above for descriptions; the common callbacks to implement are:
<pre>
  public virtual void Initialized() { }
  public virtual void Attached() { }
  public virtual void Detached() { }
  public virtual void SimulateController() { }
  public virtual void ControlGained() { }
  public virtual void ControlLost() { }
  public virtual void ExecuteCommand(Bolt.Command command, bool resetState) { }
</pre></td></tr>
<tr><td>Bolt Entity</td><td>yourEntityName_Motor.cs</td><td>Typically used by the _Controller class to do the dirty work of validating & moving a GO. Typically used methods:

<pre>
Move(input) // Returns a result struct with new position (etc) based on input.
SetState(result) // Applies the result structure to the GameObject
</pre>
Remember that <u>State</u> changes on an entity are only replicated out when changed on the Owner node for that entity.</td></tr>
<tr><td></td><td>TODO.. more?</td><td></td></tr>
</table>

4. A BoltEngine Cookbook

These recipes begin with the Bolt Tutorial

Recipe #000: Start with a stable base game (the tutorial) and check Bolt is behaving

This is important.. if you have a ‘slightly misbehaving’ install of Bolt then you will get super-confused very quickly. So always baseline with the Bolt Tutorial as described here to make sure it is working as expected before trying to add/change entities and code.

I have found the easiest way to make sure that the version of Bolt matches the version of the tutorial is to extract the tutorial from the source package. Not all bolt releases come with an updated tutorial.

GET A MAC/PC and UNITY AND OPEN THE TUTORIAL

000-02. In my tests, I started with a Mac running Unity 4.6 Pro *(Pro isn’t required, but it’s what i used). The same approach should work on Windows - just change / to \ in these instructions (and a few other things).*
000-03. Copy the bolt.unity.tutorial subfolder to a new folder and rename it bolt.tutorial.22.safecopy_001. This is the start point of your project
000-04. Open the bolt.tutorial.22.safecopy_001 folder in Unity as a Project

INSTALL BOLT. CHECK IT WORKS - SHOWS ASSETS ETC

000-05. With the project open in Unity, Double-click the Bolt_Beta_v04022_Installer.unitypackage file from Mac Finder to install Bolt. Select All, then Import
000-06. Closed Unity, then restarted it. It should opened at the same project as before.
000-07. Use the Edit menu to select INSTALL BOLT.
000-08. Quit Unity again and restart it. *(Bolt can be funny sometimes if you miss this step, idk why. Just do it)*
000-09.Use the Windows -> Bolt Engine -> ... menus to display Bolt stuff (assets, scenes, settings etc). I noted that I could see the tutorial assets for example in the *Bolt Assets* window.

MAKE THE TUTORIAL WORK

000-10. In Unity, create a new scene, call it MainMenu, save it.
000-11. In the Unity Project window, drag the bolt/scripts/BoltInit.cs script to the Main Camera in the Hierarchy window for this scene.
000-12. Save the scene again (e.g command-S)
000-13. File -> Build Settings, select "Add Current". Level1.unity was already there. I dragged them so that MainMenu was first (index 0). They do both need to be there, in that order (MainMenu:0, Level1:1)
000-14. Click "Player Settings" button.. "Full screen" off; Width=640, Height=360; "Run in background" = true; "Display Resolution Dialog" = Disabled. Now close the Build Settings window. I save for good luck again at this point (Unity.. File -> Save project)

VALIDATE THAT TUTORIAL LEVEL1 WORKS

000-15. Show the "Bolt Scenes" window (Window -> Bolt Engine -> Scenes)
000-16. Click Level1 "Debug Start"

At this point you can see the two players, run around and kill another player (right click aim, left-click shoot). Players should respawn after about 30 seconds.

Recipe #001: Creating a simple physics-controlled ball

CREATE A VERY SIMPLE NEW PREFAB FOR USE WITH BOLT REPLICATION

001-01. Open Scene "Level1"
001-02. With nothing selected, Go to Unity menu -> Game Object -> 3d -> Sphere
001-03. On the Sphere's transform, set X,Y,Z and rotations = 0. Set X,Y,Z scale = 2.
001-04. In the Unity Project window, create a new folder "recipes" at the root - this will be a peer of bolt, bolt_tutorial etc.
001-05. Select the /recipes asset folder in the Project window. It should show the folder is empty.
001-06. Drag the Sphere from the Hierarchy window to the Project window (into the /recipes folder)
001-07. Delete the Sphere from the Hierarchy window.
001-08. In the Assets->recipes folder, rename the sphere to be called TheWhiteSphere
001-09. Add a component "Bolt Entity" to TheWhiteSphere. There will be errors. it's ok..
001-10. Unity Menu: Assets -> Bolt Engine -> Update Prefab Database (this gets rid of the no id errors)
001-11. Unity Menu: Assets -> Bolt Engine -> Generate Scene Object Ids (NO IDEA IF STEP 31 WAS REQUIRED...)
001-12. Unity Menu: Assets -> Bolt Engine -> Compile assembly
001-13. Unity Window: Bolt Assets -> States... Right-click, create new State
001-14. Name it "SphereState". Change the comment to be "This is used for TheWhiteSphere"
001-15. Inheritance=Is Concrete. Bandwidth=512 .Parent: none. 16 properties/packet (all defaults)
001-16. Import Mecanim Parameters = none. (accept default)
001-17. Click the "New Property" button. Call it "Transform" and it is of type "Transform"
001-18. Set parameters.. Smoothing Algorithm = Interpolated. Axes=Position: XYZ; Rotation: XYZ. DOn't set any of the sub-parameters
001-19. Unity Menu: Assets -> Bolt Engine -> Compile Assembly.
001-20. Unity Window: Project -> recipes. Click on TheWhiteSphere to allow it to be edited
001-21. In the Inspector window, see the Serializer.. Select ISphereState as the serializer

ADD RIGIDBODY PHYSICS TO THE SPHERE AND SET IT UP FOR Transform REPLICATION

001-22. Unity Window: Project -> recipes. Click on TheWhiteSphere to allow it to be edited in the Inspector
001-23. Add Component -> Rigidbody
001-24. Unity Window: Project -> folder /bolt_tutorial/scripts/Environment
001-25. Right-click mouse to create 'TheWhiteSphere.cs'
001-26. Unity Window: Project -> Double-click TheWhiteSphere.cs in order to bring it up in the MonoDevelop editor (and ensure it is part of the C# project). There should be empty functions for Start() and Update(). Update it as follows so that only the server has the rigibody behaviors:

```
using UnityEngine;
using System.Collections;

public class TheWhiteSphere : MonoBehaviour {

	// Use this for initialization
	void Start () {
	}

	// Update is called once per frame
	void Update () {
		if (BoltNetwork.isServer == false)
		{
			// Not sure why I have to do this in Update() rather than Start().. TODO: investigate more.
			Rigidbody rb = this.GetComponent<Rigidbody>();
			rb.isKinematic = true;
			rb.useGravity = false;
		}
	}
}
```

001-27. Now, we need to make sure the transform is applied during `Attach()`. Select the TheWhiteSphere prefab and click "Add Component". Select "New Script" and call it "TheWhiteSphereController.cs". Edit the file in Monodevelop as follows:

```
using UnityEngine;
using System.Collections;

public class TheWhiteSphereController : Bolt.EntityEventListener<ISphereState>  {

      public override void Attached() {
            state.Transform.SetTransforms(transform);
      }

}
```

001-28. Unity Window: Project -> recipes -> TheWhiteSphere. In the Inspector, click "Add Component" and type TheWhiteSphere.cs. Click enter to bind that script to the prefab.

CAUSE THE SERVER TO SPAWN THE SPHERE DYNAMICALLY

001-29. Open the script bolt_tutorial\scripts\Callbacks\ServerCallbacks.cs
001-30. Change the method `SceneLoadLocalDone()` from

```
public override void SceneLoadLocalDone(string map) { // ** THIS IS THE CODE YOU ARE REPLACING **
   if (Player.serverIsPlaying)
     Player.serverPlayer.InstantiateEntity();
}
```

to be:

```
private void instantiateTheWhiteSphere()
{
  BoltEntity tws = BoltNetwork.Instantiate(BoltPrefabs.TheWhiteSphere, new TestToken(),
                                           new Vector3(0, 15f, 0), Quaternion.identity);
}

public override void SceneLoadLocalDone(string map) {  // ** THIS IS THE NEW CODE **
  if (Player.serverIsPlaying)
    Player.serverPlayer.InstantiateEntity();
  if (BoltNetwork.isServer)
    instantiateTheWhiteSphere();
}
```

001-31. (for good measure) Unity Menu: Assets -> Bolt Engine -> Compile assembly
001-32. Unity Window: Bolt Scenes -> Level1 -> Debug Start
001-33. Play now in both windows to confirm that both server and client can see and shoot at TheWhiteSphere. It is near the elevator, in the sky.

NOW, also, each player can push the sphere. Each player sees the sphere in the same place

Recipe #002: Creating a simple NPC that can take damage and die (TODO)

TODO
Pseudo: Make property on Enemy State for Health, copy over ApplyDamage function to NPCController script, on update check if health is below 1, if below 1 trigger death animation and entity.DestroyDelayed(8);

Recipe #003: Creating a drop item from a dying NPC (TODO)

TODO
Pseudo:
ServerCallbacks.cs

```
  public static void instantiateItem(Vector3 location)
  {
     BoltNetwork.Instantiate(BoltPrefabs.Item, new TestToken(), location, Quaternion.identity);
  }
```

NPCController

```
if(dead == true)
        ServerCallbacks.instantiateItem(entity.transform.position);
```

Recipe #004: Creating stuff that floats around with the player (like drones) (TODO)

TODO
Pseudo:

FlyerController.cs

```
public override void Attached()
  {
     state.Position.SetTransforms(transform);
  }
```

```
void Update()
  {
        if (entity.isOwner)
        {
        //fly around
        }
}
```

Recipe #005: A NPC with persistent state in a database (TODO)

TODO - this is just rough notes so far

In this recipe, we are using the Meteor stack since it provides a very efficient state synchronization protocol that we can leverage to efficiently sync the server with. Meteor is based on nodejs and MongoDB, and has a very similar conceptual model to Bolt, so it is a natural combination to use with Bolt (my 2c anyway), though is also 'bleeding edge'.

If just using a normal REST approach (not DDP, but quick-and-dirty to get started):

- Accessing the Database from a Unity/Bolt server: https://github.com/andyburke/UnityHTTP
- Server: meteor.com / https://github.com/awatson1978/rest-api /

Or, looking at more advanced meteor-specific DDP protocol for pub/sub between the Bolt server and Meteor:

- https://github.com/hiddenswitch/Meteor-Unity
- https://groups.google.com/forum/#!topic/meteor-talk/DcAZzwvo8EE
- http://fastchicken.co.nz/2014/02/11/making-xamarin-ios-talk-to-meteor/

Or, to be more conservative (Fholm's recommendation) - use protocol buffers and a SQL-like DB ...fholm's said.. (TODO - write up notes)

- "I would probably use a normal sql db - its what most of the big guys are using, most MMOs/hearthstone, etc."
- "I would probably have an intermediate database cache server that i would talk to over a binary tcp/ip protocol. Basically i would not have unity talk directly to a DB, I would have a binary protocol based on protocol buffers so that Unity never has to know which data store, etc. you are using"
- *Unity/C# Protobuf options.. Marc Gravell's protobuf-net*
    - *https://code.google.com/p/protobuf-net/wiki/GettingStarted*
    - *http://purdyjotut.blogspot.com/2013/10/using-protobuf-in-unity3d.html)*
    - *http://www.codeproject.com/Articles/642677/Protobuf-net-the-unofficial-manual*
    - *This is also what fholm has embedded in Bolt for serialization…*
    - *or.. https://code.google.com/p/protobuf-csharp-port/*
        - *Best instructions to follow seem to be: https://github.com/yueyoum/unity3d-scripts/tree/master/networking-with-protobuf-example*
        - *Example of socket writes to tcp-based server: https://github.com/yueyoum/unity3d-scripts/blob/master/networking-with-protobuf-example/NetWorking.cs ****
    - *or.. https://silentorbit.com/protobuf/ (No)*
        - *Github at https://github.com/hultqvist/ProtoBuf *****
        - *This is all codegen on the Unity side - no libraries*
            - *:(However.. .it uses System.Collections.Concurrent (for ConcurrentBag) which isn't in Unity, so it won't work without modification to the ConcurrentBagStack class in ProtocolParser.cs*
        - *explanatory article: http://dotdotnet.blogspot.com/2014/08/protobuf-in-unity3d.html*
    - *or.. https://github.com/pomelonode/pomelo-protobuf (No)*
        - *This one encodes the protocol in JSON and generates client-side and server-side javascript to communicate*
        -
- "it doesn't have to be protobuffers, i have written my own little protocol message system for the master server. It's also why I would not use a 'normal db' … so i can push changes from the db to the server, that is"

Building a Go-based intermediate 'database server' to use protocol buffers

(mac example.. sorry folks. I used some hints from here but simplified it a lot. Also, I'm using CSV files as the 'database' to keep it simple)

005-00. Install Go using the instructions for your platform at https://golang.org/doc/install. I used go1.4.darwin-amd64-osx10.8.pkg for my Mac
005-01. Decide where to start the project.. I'll choose ~/Code/boltrepo/
005-02. Create a folder structure to start coding the GO-based server in: ~/Code/boltrepo/servers/intermediate/golang
005-03. cd ~/Code/boltrepo/servers/intermediate/golang
005-04. Create a quick'n'dirty script that we'll use to set the go path and root for this project: create a file 'sourceme' as follows:

```
20:13:04 ~/Code/boltrepo/servers/intermediate/golang$ more sourceme
export GOPATH=`pwd`
echo GOPATH now = $GOPATH
export PATH=$PATH:$GOPATH/bin
```

005-05. 'Source' this file to set the required environment variables for Go development in this location:

```
20:17:02 ~/Code/boltrepo/servers/intermediate/golang$ . sourceme
GOPATH now = /Users/dgolds/Code/boltrepo/servers/intermediate/golang
```

005-06. Make the three main subfolders used at the root of a go project:

```
20:19:12 ~/Code/boltrepo/servers/intermediate/golang$ mkdir bin
```

```
20:19:15 ~/Code/boltrepo/servers/intermediate/golang$ mkdir pkg
20:19:20 ~/Code/boltrepo/servers/intermediate/golang$ mkdir src
```

005-07. Get the protocol buffers compiler for your platform (I used homebrew to install it). For other platforms, so the instructions at https://developers.google.com/protocol-buffers/docs/downloads

```
20:36:54 ~/Code/boltrepo/servers/intermediate/golang$ brew install protobuf
20:44:40 ~/Code/boltrepo/servers/protobuf/protobuf-2.6.1$ protoc --version
libprotoc 2.6.1
```

005-08. Get the protocol buffer libraries for the Go language from https://github.com/golang/protobuf:

```
20:48:01 ~/Code/boltrepo/servers/intermediate/golang$ echo $GOPATH
/Users/dgolds/Code/boltrepo/servers/intermediate/golang
20:48:07 ~/Code/boltrepo/servers/intermediate/golang$ go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
20:48:12 ~/Code/boltrepo/servers/intermediate/golang$ ls
bin/      pkg/      sourceme  src/
```

005-09. Create a protocol buffer file for a very simple inventory list. Call it "`GameDatabaseProtobuf.proto`" and save it in a new folder

```
20:55:22 ~/Code/boltrepo/servers/intermediate/golang/src$ mkdir GameDatabaseProtobuf
20:55:34 ~/Code/boltrepo/servers/intermediate/golang/src$ cd GameDatabaseProtobuf
20:55:40 ~/Code/boltrepo/servers/intermediate/golang/src/GameDatabaseProtobuf$ more GameDatabaseProtobuf.proto
package GameDatabaseProtobuf;

message FullInventory {
  required int32 clientId = 1;
  repeated InventoryItem InventoryItems = 2;

   message InventoryItem {
    required int32 ItemId = 3;
    required int32 Quantity = 4;
   }
}
```

005-10. Compile and build the protocol buffer file:

```
23:48:18 ~/Code/boltrepo/servers/intermediate/golang/src/GameDatabaseProtobuf$ protoc --go_out=.
GameDatabaseProtobuf.proto
23:48:23 ~/Code/boltrepo/servers/intermediate/golang/src/GameDatabaseProtobuf$ ls
GameDatabaseProtobuf.pb.go  GameDatabaseProtobuf.proto
23:48:29 ~/Code/boltrepo/servers/intermediate/golang/src/GameDatabaseProtobuf$ go build
23:48:37 ~/Code/boltrepo/servers/intermediate/golang/src/GameDatabaseProtobuf$ cd ..
```

005-11. Create a very simple protocol buffer server (this will just save as a CSV for now):

```
23:52:11 ~/Code/boltrepo/servers/intermediate/golang/src$ mkdir GoProtocolServer
23:52:13 ~/Code/boltrepo/servers/intermediate/golang/src$ cd GoProtocolServer
23:52:17 ~/Code/boltrepo/servers/intermediate/golang/src/GoProtoServer$ vi GoProtocolServer.go

package main

import (
  "fmt"
  "github.com/golang/protobuf/proto"
  "net"
  "os"
  "GameDatabaseProtobuf"
  "encoding/csv"
  "strconv"
  )

  func main() {
    fmt.Printf("Started ProtoBuf Server")
    c := make(chan *GameDatabaseProtobuf.FullInventory)
    go func(){
      for {
        message := <-c
        writeValuesTofile(message)

      }
      }()
      // Listen to the TCP port
      listener, err := net.Listen("tcp", "127.0.0.1:2110")
      checkError(err)
      for{
        if conn, err := listener.Accept(); err == nil{
          // If err is nil then that means that data is available for us so we take up this data and pass it to a new goroutine
          go handleProtoClient(conn, c)
          } else{
```

```
      continue
    }
  }
}

func handleProtoClient(conn net.Conn, c chan *GameDatabaseProtobuf.FullInventory){
  fmt.Println("Connection established")
  // Close the connection when the function exits
  defer conn.Close()
  // Create a data buffer of type byte slice with capacity of 4096
  data := make([]byte, 4096)
  // Read the data waiting on the connection and put it in the data buffer
  n,err:= conn.Read(data)
  checkError(err)
  fmt.Println("Decoding Protobuf message")
  // Create an struct pointer of type ProtobufTest.TestMessage struct
  protodata := new(GameDatabaseProtobuf.FullInventory)
  // Convert all the data retrieved into the ProtobufTest.TestMessage struct type
  err = proto.Unmarshal(data[0:n], protodata)
  checkError(err)
  // Push the protobuf message into a channel
  c <- protodata
}

func writeValuesTofile(datatowrite *GameDatabaseProtobuf.FullInventory){

  // Retrieve client information from the protobuf message
  ClientID := int(datatowrite.GetClientId())
        // retrieve the message items list
        items := datatowrite.GetInventoryItems()
        fmt.Println("Writing value to CSV file")
        // Open file for writes, if the file does not exist then create it
        file,err := os.OpenFile("CSVValues.csv", os.O_RDWR|os.O_APPEND|os.O_CREATE, 0666)
        checkError(err)
        // make sure the file gets closed once the function exists
        defer file.Close()
        // Go through the list of message items, insert them into a string array then write them to the CSV file.
        writer := csv.NewWriter(file)
        for _,item := range items{
          record := []string{strconv.Itoa(ClientID), strconv.Itoa(int(item.GetItemId())),
strconv.Itoa(int(item.GetQuantity()))}
          writer.Write(record)
          fmt.Println(record)
        }
        // flush data to the CSV file
        writer.Flush()
        fmt.Println("Finished Writing value to CSV file")
      }

      func checkError(err error){
        if err != nil {
          fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
          os.Exit(1)
        }
      }

23:52:42 ~/Code/boltrepo/servers/intermediate/golang/src/GoProtoServer$ go build
23:53:11 ~/Code/boltrepo/servers/intermediate/golang/src/GoProtoServer$ go install
23:53:12 ~/Code/boltrepo/servers/intermediate/golang/src/GoProtoServer$ ls ../../bin
GoProtoServer* protoc-gen-go*
23:53:23 ~/Code/boltrepo/servers/intermediate/golang/src/GoProtoServer$ GoProtoServer
Started ProtoBuf Server
```

At this point we have a working, but very simple Protocol Buffer server written in Go, and just writing to CSV files. See this article to get an understanding of how this intermediate server works

NEXT: IMPLEMENT THE SAME PROTOCOL BUFFER CODE ON UNITY SO THE BOLT SERVER CAN READ/WRITE DB via PROTOBUFs

Building a protocol buffer client in Unity to connect to the 'Game DB Server'

005-12. Bring in a Unity protocol buffer library

TODO - finish this recipe on the Unity side

Recipe #006: Customized Bolt Settings configuration at runtime (TODO)

TODO reference https://github.com/BoltEngine/bolt_documentation/blob/master/wiki/CustomConfiguration.md and http://wiki.boltengine.com/wiki/4/custom-configuration

Recipe #007: Interpolating / DR for less jitter (TODO)

TODO -See https://github.com/BoltEngine/bolt_documentation/blob/master/wiki/InterpolatedSnapshots_vs_DeadReckoning.md and http://wiki.boltengine.com/wiki/5/interpolation-vs-extrapolation

Note that there have been jitter issues with Interpolation for a few releases and they were finally fixed in 0.4.1.5, so do update to 0.4.1.5 if you are having issues with interpolation.

Recipe #008: Inventory for a Player (TODO)

TODO - Use the State replication destinations to only replicate to **Controller. A good approach is to simply have a Inventory structure on the Player, but set the Inventory structure to only replicate between owner and controller.**

5. Appendix

Information Sources used in this document

- The Reddit Bolt 0.4 Wiki (stale)
- http://doc-api.photonengine.com/en/bolt/current/index.html
- The Slack (Jabbr archive is pinned)
- The Bolt Forum

To Do's to update this doc

- .SetDynamic and .GetDynamic (a way of getting/setting states from other assets...)
- change from mod = State.Modify() to direct setters/getters (removed in 0.4.3)
- In 0.4.1.5 and forward you can also call entity.Freeze(true); to stop it from updating completely (note that Bolt now has SetIdle, Freeze, Attach/Detach and Priority - need to do a write up here or refer to fholms of he does one).
- new 'attach on load' flag in the scene properties of "Bolt Entity" " you can modify it on the scene object"
- POI and AOI
    - Bolt's AOI stuff does instantiate/destroy: https://www.youtube.com/watch?v=ELp702pRNg4&spfreload=10
    - Reference this in with the 'don't disable game object stuff'

Game genres that BOLT is not appropriate for

a) Don't use BOLT for games that require deterministic multiplayer simulation (e.g. RTS)

See for an intro on lockstepping. Bolt can't run the simulation for the same reason Unity can't run the simulation:

For example in Starcraft 1 and 2, they don't sync unit positions, or health, or ANYTHING over the network, the only thing they sync is keypresses + mouse clicks, aka user input. The code runs the simulation locally for each person, with everyone's kb/mouse input and the output is identical on each machine. Basically determinism is about mathematical purity, the entire simulation can be described like this: s' = f(s) where s is the current state, s' is the next state, and the simulation is the function f() and there are no more parameters to the function

In fact the non-determinism (ordering etc) of Unity's MonoBehavior callbacks means much of the logic cannot be in Unity callbacks (Update(), FixedUpdate(), no co-routines, Awake(), Start() etc).. So Unity can be used as nothing more than a presentation of a game that is simulated outside of Unity (e..g in a DLL). Similarly, since Bolt runs in Unity callbacks, it can't provide the determinism either without a major rewrite (and would be a very different thing).

Consider something like this https://www.photonengine.com/en-US/TrueSync

8. Troubleshooting resources

1. Grok this document!
2. https://doc.photonengine.com/en-us/bolt/current/reference/troubleshooting
3. The Slack (get invite here https://photonbolt.herokuapp.com/)

Quick list of recent/common issues:

- The POI/AOI examples provided with Bolt will not work with scene entities