***Development Guidelines***

# 

Table of Contents Page

[Preface 4](#_Toc52310029)

[C# Coding Conventions 5](#_Toc52310030)

[Variable Declarations 5](#_Toc52310031)

[*Prefix private variables with an underscore* 5](#_Toc52310032)

[*Declare variables on separate lines* 6](#_Toc52310033)

[*Avoid using the var keyword* 7](#_Toc52310034)

[*Group related variables together* 8](#_Toc52310035)

[*Use camelCase for variable names* 10](#_Toc52310036)

[Method Declarations 11](#_Toc52310037)

[*Add accessibility modifiers to all methods for clarity* 11](#_Toc52310038)

[*Use PascalCase for methods* 11](#_Toc52310039)

[*Order methods in terms of their modifiers, then their accessibility* 11](#_Toc52310040)

[Property Declarations 14](#_Toc52310041)

[*Use PascalCase for properties* 14](#_Toc52310042)

[*Order auto-properties after all variables and in terms of accessibility* 14](#_Toc52310043)

[*Order method-like properties after auto-properties* 15](#_Toc52310044)

[*Place properties with backing variables next to their variables* 16](#_Toc52310045)

[Delegate Declarations 17](#_Toc52310046)

[*Declare all delegates after properties and before events* 17](#_Toc52310047)

[*Arrange delegates with regards to accessibility* 17](#_Toc52310048)

[*Write your own delegates than using System.Action and System.Func* 17](#_Toc52310049)

[C# Event Declaration 19](#_Toc52310050)

[*Declare all events after delegates and before Unity methods* 19](#_Toc52310051)

[*Arrange events in terms of their accessibility* 19](#_Toc52310052)

[UnityEvent Declaration 20](#_Toc52310053)

[*Declare UnityEvent fields at the end of their related group of variables* 20](#_Toc52310054)

[*Declare child classes for UnityEvent<T0..T3> at the top of the class* 22](#_Toc52310055)

[*Use "Event" suffix for child classes derived from UnityEvent<T0..T3>* 23](#_Toc52310056)

[*Use PascalCase and an "On" prefix for UnityEvent field names* 24](#_Toc52310057)

[Summary of Ordering and Conventions 26](#_Toc52310058)

[Naming Conventions 29](#_Toc52310059)

[*Branches* 29](#_Toc52310060)

[*Tags* 29](#_Toc52310061)

[Commit Messages 29](#_Toc52310062)

[PHP Coding Conventions 30](#_Toc52310063)

[*Programming Paradigm* 30](#_Toc52310064)

[*Naming conventions* 30](#_Toc52310065)

[*Ordering* 31](#_Toc52310066)

[XAMPP 31](#_Toc52310067)

[Database Schema 32](#_Toc52310068)

[*Schema* 32](#_Toc52310069)

# Preface

This document is a handbook comprising project conventions and server-side set-up deployed in the development of Project <XXX>. Aspects covered by this document are conventions used in coding and the version control system used to sync project progress between members.

# C# Coding Conventions

The following lists all the conventions we've set up for the C# programming language in our project.

Most of the coding conventions are enforced using a file known as a .editorconfig (read more [here](https://editorconfig.org/)). This file is placed in the root of the Git repository, allowing projects to make use of the coding conventions defined in the file. These conventions specify the syntax of the code, such as whether the "this" keyword is preferred and how the curly braces are placed.

These coding conventions are generated using Visual Studio 2019 (read more [here](https://docs.microsoft.com/en-us/visualstudio/ide/code-styles-and-code-cleanup?view=vs-2019#code-styles-in-editorconfig-files)).

However, there are some conventions that have to be instilled within the mindsets of our developers, since these conventions cannot be enforced using a .editorconfig file. Examples include how fields, properties and methods are sorted. Hence, these conventions must be done by the developers themselves.

# Variable Declarations

## Prefix private variables with an underscore

All private variables are to be prefixed with an underscore.

|  |
| --- |
| private float \_calculatedDamage; |

The rationale being it makes it clear without having to find the declaration of the variable.

If the variable is serialized using SerializeField, it shows up in Unity without the underscore prefix.

## Declare variables on separate lines

Avoid declaring variables in the same line. Instead, declare them on multiple lines.

|  |
| --- |
| // Prefer public int currentNumberOfCycles; public int maxNumberOfCycles;  // Avoid public int currentNumberOfCycles, maxNumberOfCycles;  // Avoid public int currentNumberOfCycles; public int maxNumberOfCycles; |

The rationale being that such declarations take up more horizontal space and are more difficult to comment.

Consider the following scenario on commenting:

|  |
| --- |
| public float damage, resistance, armour; |

How would you go about commenting about what these variables do?

One way would be to do this:

|  |
| --- |
| // health - number of hits before death // resistance - the chance of preventing health from reducing // armour - reduce damage received by this percentage public float health, resistance, armour; |

The downside is that not every line of the comment corresponds to the correct variable. For instance, the first line of comment on the health variable does not correspond to the other variables because it is not related to them in any way. This may make documentation harder to read.

An alternative would be to separate each variable on its own line and have a line of the comment above it:

|  |
| --- |
| // number of hits before death public float health;  // reduce damage received by this percentage public float resistance;  // chance of preventing health from reducing public float armour; |

Notice how we don't have to identify what variable we are describing in the comment because it is clear that we are describing the variable below the line of comment, so one can simply see the description by looking at the line above.

## Avoid using the var keyword

Consider using an explicit type (e.g. string, int, GameObject) instead of letting the compiler infer from the type using var.

|  |
| --- |
| // Prefer public string name = "John";  // Avoid public var name = "John"; |

The var keyword has been a highly controversial topic. Some love it, some hate it. Some say it improves readability and cleanliness, others say it makes it harder to see the type of the variable.

We currently see many downsides with using the var keyword. You can read the remarks section of this Microsoft document [here](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables#remarks) to see the restrictions of using the var keyword.

One big driver of not using the var keyword is when the return type is not apparent from a method's name.

Consider the following code snippet:

|  |
| --- |
| var names = GetNames();  names.Add("John"); |

What might be the return type for the GetNames method? It might be an array, a list or even a string.

If the return type for the GetNames is an array, the code will not compile because there's no such Add method on an array.

We currently doubt that Unity's Scripting API would have super long type names, which is another reason why we are not using the var keyword.

However, as times go on, we may have to revise this section. Do stay tuned to any changes that may take place in this section.

## Group related variables together

When possible, consider placing related variables together on separate lines. Each group should be separated with a newline.

|  |
| --- |
| public float damageAmount; public float damageFrequency; public float damageCooldown;  public float moveSpeed; public float turnSpeed; |

This allows the use of Unity's Header attribute, which places a header on top of the variable the attribute is on.

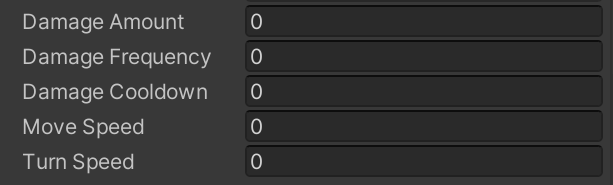
Consider the following code snippet:

|  |
| --- |
| [Header("Damage")] public float damageAmount; public float damageFrequency; public float damageCooldown;  [Header("Motion")] public float moveSpeed; public float turnSpeed; |

With Header attributes, this is how the variables look like in the Inspector window of Unity:



Without Header attributes, this is how the variables look like in the Inspector window:



Notice that the addition of the Header attribute adds a header above the variable it is placed on. It also allocates some space above it so that other variables don't clutter it, clearly separating the two groups.

## Use camelCase for variable names

This is the default convention for cases in variables. Hence, stating why we chose to stick with it.

|  |
| --- |
| public float volumeLevel; // ok public float VolumeLevel; // not ok public float VOLUME\_LEVEL; // not ok public float VoLuMeLeVeL; // not ok - consult your psychiatrist ;) |

# Method Declarations

## Add accessibility modifiers to all methods for clarity

Specifying the accessibility modifier makes it clear what can access the method.

|  |
| --- |
| public float CalculateDamage() { //... }; protected float NormalizeValues() { //... }; private void Update() { //... }; |

If no accessibility modifier is specified, C# will provide a default accessibility level as described [here](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/accessibility-levels).

The rationale here is that you can easily see the accessibility without having to memorize the default accessibility level of the method. Furthermore, classes and interfaces in C# have a different default level of accessibility (private for classes and public for interfaces).

## Use PascalCase for methods

This is the default convention for cases in methods. So, we will stick with it.

|  |
| --- |
| public float GetComputedRate() { ... }; // ok public float getComputedRate() { ... }; // not ok public float get\_computed\_rate() { ... }; // not ok public float GET\_COMPUTED\_RATE() { ... }; // not ok |

## Order methods in terms of their modifiers, then their accessibility

We order the methods in terms of what modifiers are given to it. To make it easy, we will split the methods group by group, then order them within their group.

The three groups are named and ordered like so:

1. Abstract methods
2. Overridden methods
3. Unity methods (a.k.a. Unity messages, like Update)
4. Virtual methods
5. Standard methods

Within the three groups, we order the methods according to their accessibility level. The more accessible the method is, the more it should be declared above after methods.

We do not order methods alphabetically because it will make refactoring much harder due to the need to re-arrange contents. Given that we are using a version control system, there is a higher chance of a merge conflict because we are moving lots of content around.

We will be referring to [this Microsoft document](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/accessibility-levels) to determine the accessibility level of each accessibility modifier, and hence use the information to order methods in terms of accessibility.

For the most part, internal, protected internal and private protected can be ignored since we will be working within Unity's assembly.

|  |
| --- |
| // Abstract methods are first public abstract float GetCalculatedCost(); protected abstract void DealDamage(float damage);  // Overridden methods are next public override int GetID() { ... }; protected override string GetName() { ... };  // Then Unity methods private void Update() { ... }; private void OnCollisionEnter() { ... };  // And next is virtual methods public virtual void SendItem(Item item) { ... }; public virtual float CalculateValue() { ... };  // And lastly, standard methods public void SendEnemy() { ... }; protected int GetRoundedValue() { ... }; private float CalculateRate() { ... }; |

We set to order abstract methods first because these methods are required to be implemented by its inherited classes and it is important to pay attention to the method signature and return type of the method.

We order overridden methods next to make it clear to the developer of which methods are being overridden from the base class.

Next, we place Unity methods after overridden methods since these are special methods that Unity will call.

We place virtual methods after Unity methods because of its significance. There is a reason why a method is marked as virtual, which allows inherited classes to override it and hence is why we want to put it above standard methods.

Lastly, we place standard methods. We place them last because of their lack of modifiers, except for their accessibility modifier.

# Property Declarations

## Use PascalCase for properties

This is the default convention for cases in properties.

|  |
| --- |
| public int TotalCost { get; set; } // ok public int totalCost { get; set; } // not ok public int TOTAL\_COST { get; set; } // not ok public int total\_cost { get; set; } // not ok |

## Order auto-properties after all variables and in terms of accessibility

All auto-properties are to be ordered right after all variables and they are ordered in terms of their accessibility. Accessibilities given to the get and set parts do not count.

The rationale behind this is that auto-properties themself already create a private backing field that can only be accessed via the property, which is safe to make them independent and standalone from the rest of the group. Read [here](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/auto-implemented-properties) for more information.

|  |
| --- |
| // variables public int monetaryValue = 32; public float force = 1f;  // auto properties public string Name { get; private set; } protected GameObject Target { get; set; } |

A possible good use case of auto-properties might be using classes where you have a public variable that you do not want Unity to serialize since Unity cannot serialize properties. You can also use auto-properties to fine-tune the get and set accessibility to limit who can read or modify the property.

## Order method-like properties after auto-properties

Method-like properties are properties very similar to methods where accessing and mutating them are like doing method calls.

Here is an example of one:

|  |
| --- |
| public int Sum {  get  {  return a + b;  }   // you can also do a "set" here so that code will be  // executed when Sum is mutated  // you can access the value assigned  // via the "value" keyword } |

Within the group, be sure to order method-like properties in terms of the accessibility used in its declaration.

The rationale is that such properties are "method-like", so they can be placed near methods. Also, we keep the accessibility order to keep it consistent with how we order the others.

## Place properties with backing variables next to their variables

Sometimes, you need Unity to serialize a variable, but do not want it to be publicly accessed by other scripts and instead, through a property. To accomplish this, you can use a property that accesses and mutates the private variable, like so:

|  |
| --- |
| [SerializeField] private GameObject target;  public GameObject Target {  get  {  return target;  }    set  {  target = value;  } } |

Since these properties are meant to provide a way to access and mutate the private variable, they should be placed after their variables, separated by one newline.

# Delegate Declarations

## Declare all delegates after properties and before events

Delegates are most likely used by events, so they are declared before events to make it clear of its presence.

|  |
| --- |
| public delegate void ValueChangedDelegate(float newValue);  public event ValueChangedDelegate ValueChangedEvent; |

## Arrange delegates with regards to accessibility

The most accessible delegates are placed at the top of the group while the least accessible delegates are placed at the bottom of the group.

|  |
| --- |
| public delegate void MyDelegate1(); protected delegate void MyDelegate2(); private delegate void MyDelegate3(); |

## Write your own delegates than using System.Action and System.Func

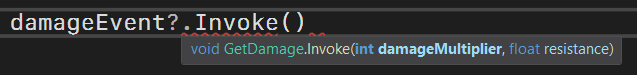
Although the System.Action and System.Func delegates are well-known, there are some downsides to it.

Firstly, you cannot use out and ref keywords in System.Action and System.Func compared to standard delegates, as these will cause a compile-time error.

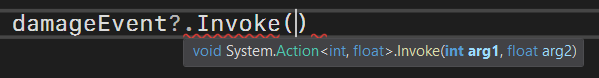
|  |
| --- |
| // Using System.Action vs normal delegates  // Both delegates below will compile just fine private delegate void PerformSomething(out int value); private delegate void DoSomethingElse(ref int value);  // These, however, cannot compile  // CS1960 - Invalid variance modifier System.Action<out int> PerformSomethingAction;  // CS1073 - Unexpected token 'ref' System.Action<ref int> DoSomethingElseAction; |

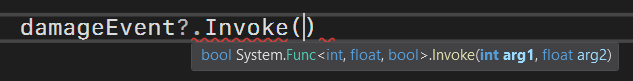
|  |
| --- |
| // Using System.Func vs normal delegates // Both delegates below will compile just fine private delegate int GetAmount(out int value); private delegate int SetAmount(ref int value);  // These however, cannot compile // CS1960 - Invalid variance modifier System.Func<out int, int> GetAmountFunc;  // CS1073 - Unexpected token 'ref' System.Func<ref int, int> SetAmountFunc; |

Furthermore, if you use IntelliSense with delegates, you get the names of the arguments, such as when invoking an event.



If you use it with System.Action or System.Func, you will not get a very descriptive name for its arguments.





# C# Event Declaration

## Declare all events after delegates and before Unity methods

Most likely a delegate declared will be used by an event, so all events are to be placed right after delegates are declared and before Unity methods.

|  |
| --- |
| public delegate void DoSomethingDelegate();  public event DoSomethingDelegate DoSomethingEvent;  private void Awake() { } |

Unlike UnityEvent, C# events are not shown in the Inspector within Unity, so to make it easier to access them, it will be better to place them together.

## Arrange events in terms of their accessibility

Within the group, arrange events such that the most accessible event is at the top and the least accessible event is at the bottom.

|  |
| --- |
| public event MyDelegate1 MyEvent1; protected event MyDelegate2 MyEvent2; private event MyDelegate3 MyEvent3; |

# 

# [UnityEvent](https://docs.unity3d.com/Manual/UnityEvents.html) Declaration

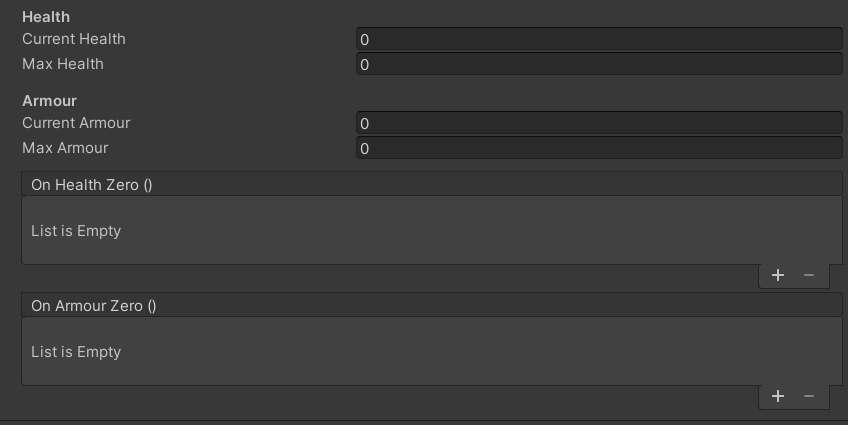
## Declare UnityEvent fields at the end of their related group of variables

Placing UnityEvents to their related groups helps with readability of both the code and in the Inspector window in Unity (e.g. health fields all in one place, armour fields all in another place).

Consider the following example, where the UnityEvents are not grouped together with other variables:

|  |
| --- |
| [Header("Health")] public float currentHealth; public float maxHealth;  [Header("Armour")] public float currentArmour; public float maxArmour;  // for the sake of keeping things looking neat  // and not cluttered [Space] public UnityEvent OnHealthZero; public UnityEvent OnArmourZero; |

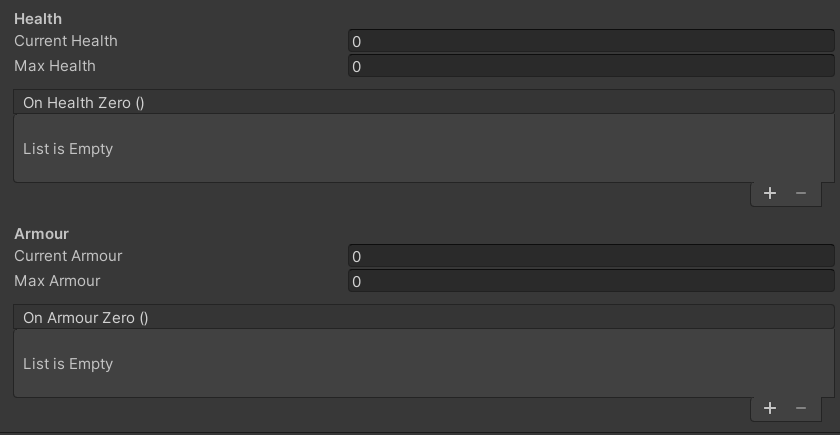
This is how a script with these fields looks like in the Inspector window.



Now, the UnityEvents are arranged so that they are in the group they are related:

|  |
| --- |
| // we're adding [Space] to make things neat  [Header("Health")] public float currentHealth; public float maxHealth; [Space] public UnityEvent OnHealthZero;  [Header("Armour")] public float currentArmour; public float maxArmour; [Space] public UnityEvent OnArmourZero; |

And this is how the script looks like:



## Declare child classes for UnityEvent<T0..T3> at the top of the class

Due to how Unity works, generic versions of UnityEvent must be inherited with a serializable class to have it show up in the Inspector. See [this page](https://docs.unity3d.com/ScriptReference/Events.UnityEvent_1.html) on the documentation for UnityEvent<T0> (the same applies for generic UnityEvents with 2 or more type arguments).

Since the definition of the serializable class doesn't have any implementation, it can be easily represented in one line and it is better to put it at the top of the class to make navigation easier.

|  |
| --- |
| public class Health : MonoBehaviour {  [System.Serializable] public class HealthChangedEvent : UnityEvent<int> { }  [System.Serializable] public class HealthZeroEvent : UnityEvent<int> { }   public HealthChangedEvent OnHealthChanged;  public HealthZeroEvent OnHealthZero; } |

## Use "Event" suffix for child classes derived from UnityEvent<T0..T3>

Use the word "Event" after the name of the class to better indicate that it is an event (e.g. EnemyDamaged**Event**).

|  |
| --- |
| // not ok  [Serializable] public class PlayerDamaged : UnityEvent<int> { }  // ok  [Serializable] public class PlayerDamagedEvent : UnityEvent<int> { }  // not ok - place Event at the end, not at the front  [Serializable] public class EventPlayerDamaged : UnityEvent<int> { } |

The rationale behind this is to make it easier to identify UnityEvent-derived classes from other classes.

Consider the following code:

|  |
| --- |
| [Serializable] public class PlayerDamaged : UnityEvent<int> { } [Serializable] public struct PlayerDamage {  public float amount; }  // looking at ONLY these 2, which one is the UnityEvent? public PlayerDamaged playerDamaged; public PlayerDamage playerDamage; |

In the code, it is hard to tell which field is a UnityEvent until you see the declaration of it.

Now, consider the following code, where we add an "Event" suffix to the UnityEvent-derived class:

|  |
| --- |
| [Serializable] public class PlayerDamagedEvent : UnityEvent<int> { } [Serializable] public struct PlayerDamage {  public float amount; }  // looking at ONLY these 2, which one is the UnityEvent? public PlayerDamagedEvent playerDamaged; public PlayerDamage playerDamage; |

It should now be easy to see that PlayerDamagedEvent is the UnityEvent just by looking at it.

## Use PascalCase and an "On" prefix for UnityEvent field names

Declare UnityEvent (including their derived classes) field names with an "On" prefix and using PascalCase.

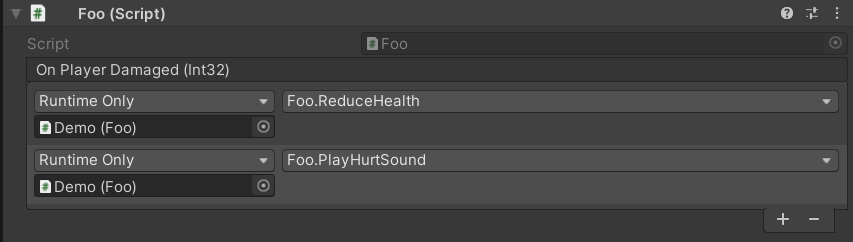
|  |
| --- |
| public UnityEvent OnPlayerDamaged; // ok public UnityEvent onPlayerDamaged; // not ok public UnityEvent PlayerDamaged; // not ok  public UnityEvent playerDamaged; // not ok |

The rationale behind this is to keep them consistent with the names that Unity gives for their in-game UI controls like buttons and sliders. The names are OnClick and OnValueChanged respectively.

While C# treats UnityEvents like regular fields, they are inherently events that can be invoked and have listeners; it'd be better to treat such fields as events. Since C# events use PascalCase, we'd follow this trend and let UnityEvents use PascalCase.

Using the "On" prefix allows the event to convey a meaning to users.

Consider the following:



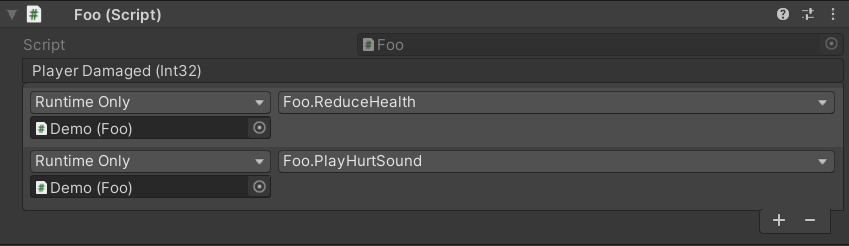
Above, it is easy to tell that when the player is damaged, the health of the player will be reduced and a hurt sound will be played.

The event above attempts to convey the following meaning:

*"On player being damaged, reduce health and play hurt sound"*

While the meaning is not grammatically correct, there is a clear consequence of what happens when the player is damaged.

Consider another example, which doesn't use the "On" prefix:



It might still be easy to tell, but the meaning conveyed might be a little confusing and it may take a while to understand the intentions of this event.

The event above conveys the following meaning:

*"Player being damaged, reduce health and play hurt sound"*

Also, not grammatically correct, but when read, it may feel confusing because it sounds more like 3 actions being executed and not 2 actions being executed on account of the first one.

# Summary of Ordering and Conventions

The following shows the ordering and conventions of all the items mentioned in this document, in one script.

|  |
| --- |
| using System; using System.Collections.Generic; using UnityEngine;  public class Ordering : MonoBehaviour {  // classes derived from UnityEvent for the sake of making  // the UnityEvent serializable are placed at the very top  // and these are arranged according to accessibility  [Serializable] public class PlayerDamagedEvent : UnityEvent<int> { }  [Serializable] public class PlayerDiedEvent : UnityEvent<int> { }  [Serializable] protected class MyEvent : UnityEvent<float> { }  [Serializable] private class MyEvent2 : UnityEvent<float> { }   // don't care about accessibility in these fields  // if they are serialized, do consider their arrangement   // since they will show up in the Inspector  [Header("Player Health")]  public float currentHealth;  public float maxHealth;  public PlayerDamagedEvent OnPlayerDamaged;  public PlayerDiedEvent OnPlayerDied;  // more code shown on next page...  [Header("Damage")]  [SerializeField] protected float damage;  private float \_internalDamage;  // the property below uses "damage" as the backing variable  public float Damage { get => damage; set => damage = value; }   // auto properties  public float Currency { get; private set; }  protected float Income { get; set; }  private int Id { get; set; }   // method-like properties, in terms of accessibility  public float CurrencyIncomeSum { get => Currency + Income; }  protected float Product { get => Currency \* Income; }  private float Quotient { get => Currency / Income; }  // delegates are all declared here  // they are arranged in terms of accessibility  public delegate void MyDelegate1();  protected delegate void MyDelegate2();  private delegate void MyDelegate3();   // events are all declared next  // they are also arranged in terms of accessibility  public event MyDelegate1 MyEvent1;  protected event MyDelegate2 MyEvent2;  private event MyDelegate3 MyEvent3;   // abstract methods are after events  // arranged in terms of accessibility  public abstract void AbstractAction1();  protected abstract void AbstractAction2();   // overridden methods after abstract ones  // arranged in terms of accessibility  public override void OverriddenMethod1() { }  protected override void OverriddenMethod2() { }   // methods used by Unity are all here  // also order them in terms of accessibility  // which shouldn't be a problem if you use private all the way  private void Awake() { }  private void Start() { }  private void Update() { }   // virtual methods here  // in terms of accessibility  public virtual void VirtualMethod1() { }  protected virtual void VirtualMethod2() { }  // now, standard methods  // in terms of accessibility  public void YourNormalMethod1() { }  protected void YourNormalMethod2() { }  private void YourNormalMethod3() { } } |

Version Control System

Git version control will be utilized for controlling development updates and synchronization between team members for Project <XXX>. The git-flow methodology will be used in particular to streamline the synchronization process.

# 

# Naming Conventions

Listed are naming conventions for the various branch types in the git-flow methodology.

Names comprising two or more words should be compounded with a hyphen*(‘-’).* No whitespace should be used (*e.g. this-is-my-branch*).

## Branches

**Feature branches** should start with ‘feature/’ followed by the feature name *(e.g feature/a-feature)*

## Tags

**Version tags** should be prefixed with ‘v’ *(lower case) followed by the release version* *(e.g v0.1, v2.1)*

# Commit Messages

This article is all you need. Yes.

<https://chris.beams.io/posts/git-commit/>

Server-side Development

This section shall go over setting-up XAMPP for a local Apache server testing and development environment. Basic coding conventions for our main server-side scripting language, PHP, will also be covered. Additionally, steps for setting up Hamachi to test LAN over WAN is included for ease of testing with other developers.

# 

# PHP Coding Conventions

The server is scripted with PHP, which is relatively liberal in coding conventions when compared to other languages. Nevertheless, there are some basic naming rules and consistent paradigms used when scripting on the server to make scripts clearer and simplify handing over to future developers.

## Programming Paradigm

PHP supports many conventional programming paradigms such as object oriented, functional, and procedural. We have chosen to use procedural programming due to our small server-side codebase and faster development iteration speeds. We may consider using an object-oriented style if our codebase grows larger and procedural programming proves difficult to manage.

## Naming conventions

**Script files** are in snake case *(e.g. test\_script.php, unity\_auth.php)*

**Functions** are in snake case *(e.g. do\_something(), test\_func())*

**Variables** are in snake case *(e.g, $value, $int\_variable)*

**Constants** are in capital case and separated by underscores

*(e.g. define(“MY\_CONSTANT”, 10), define(“AN\_ARRAY”, [“value”]))*

## Ordering

**Functions** are placed at the bottom of the script

**Variables** are declared and initialized when needed anywhere in the file

**Constants** are declared at the top of the script

# 

# XAMPP

XAMPP is a useful Apache distribution to locally test server-side scripting logic. It is used widely in Project FAST for testing php scripts for the server and database management.

XAMPP can be downloaded here:  
<https://www.apachefriends.org/index.html>

To set up the server scripts, just drag and drop the server scripts in the htdocs folder included in the source package into your local XAMPP htdocs folder. The server development environment consists of htdocs as the root of all server-side scripts.

# 

# Database Schema

An SQL database is run on the server to provide network storage and authentication services to Project <XXX>.

***Note :***

*The project convention establishes that all SQL element names are in snake case (e.g. project\_fast\_db, session\_record)*

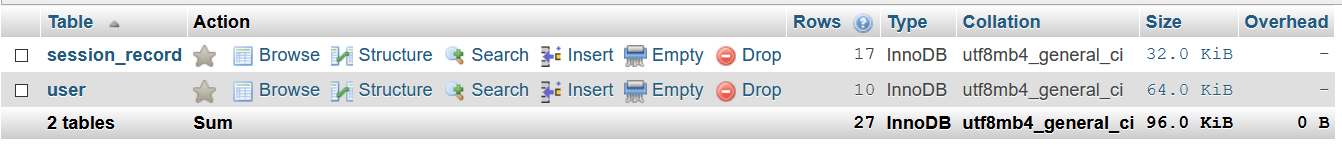
## Schema

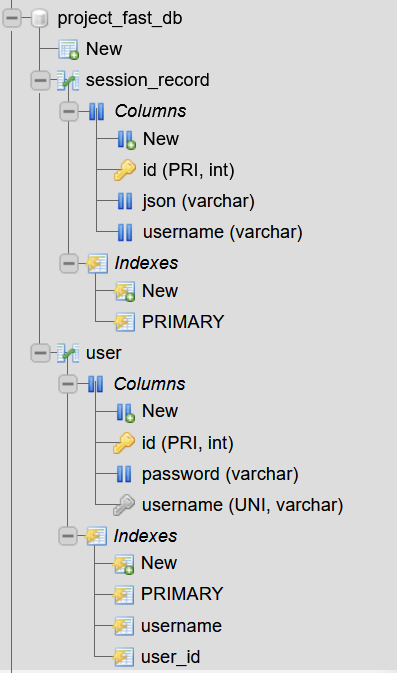
The database schema is described in this section.

Database Name : project\_fast\_db

Tables

* user
* session\_record



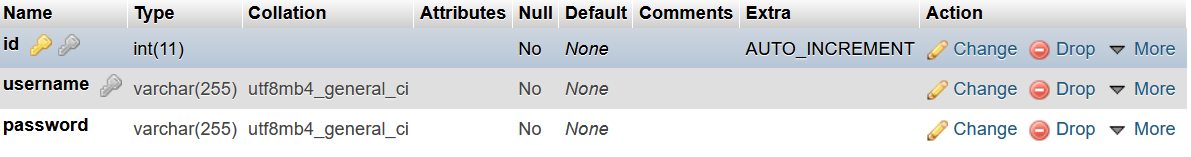


User Table

id (int) - Primary key, unique, not null, auto-increment

username (varchar 255) - unique, not null

password (varchar 255) - not null



Session Record table

id (int) - Primary key, not null, auto-increment

json (varchar 512) - not null

username (varchar 255) - not null

