

Sketch Dialogue

Dialogues template

Documentation for Unity3D

Version: 1.0
Author: Perfect Human Apps
Last Updated: 02 May 2021
Web Site: Perfect-Human.com

Thank you for purchasing and using the Sketch Dialogue template!

This template allows you to implement a dialogue module between the player and game characters in your project. In addition, the functionality of the template allows you to directly influence the characteristics of the character based on the results of the player's decisions.

The front-end is made in C # (in the Unity3D environment), the server back-end works in the MySQL + PHP bundle. Client-Server architecture is optional, all scripts can be described inside the code for a completely stand-alone application or downloaded from your server on demand. Important! If you plan to use a client-server architecture, you will need basic knowledge of client-server communication. In addition, knowledge of MySQL and PHP is highly desirable for fine-tuning the template. This documentation is provided with a detailed guide to each stage of work both on the front-end (in Unity) and on the back-end (PHP scripts and binding with the MySQL server). However, such steps as deploying your own server (on a local machine or hosting), registering and / or obtaining an authorization token on partner sites will be mentioned only in passing, with links to resources where such steps are discussed in more detail. In any case, you must clearly understand what data and where you will receive to form your set of scenarios.

In case of any difficulties, you can contact me. All contacts are listed in the "Feedback and Support" section.

Please provide feedback on the Asset Store. Your feedback is extremely important!

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ	2
1. OVERVIEW OF COMPONENTS	3
1.2 A set of base control classes.	3
1.3 Server-side PHP scripts for interacting with Unity.....	4
2. CONFIGURING CLIENT MODULES.....	5
2.1 Data storage and retrieval module.	5
2.1.1 Main functions.	5
2.2 Scenario scripting module.....	7
2.2.1 Processing the stages of the script.	7
2.2.2 Apply milestone conditions, replicas, and feature generation.....	7
2.2.3 Handling special events.....	8
2.2.4 Accrual of characteristics.	8
2.3 Additional modules.	9
2.3.1 Avatar selection module.	9
2.3.2 Choosing a name.....	10
2.3.3 Gender selection.	11
2.4 Programmatic scenario input.....	12
2.4.1 Description of the demo script.	12
2.4.2 Creation of script characters.....	12
2.4.3 Entering the stages of the script.	12
2.4.4 Adding replicas to stages.	13
3. CONFIGURING SERVER MODULES.....	14
3.1 General information for working with hosting.	14
3.1.1 Buying hosting and getting data for work.....	14
3.1.2 Writing script files to the server through ISP panel.....	14
3.1.3 Importing database into phpMyAdmin.....	15
3.1.4 PHP version and required libraries.	16
3.1.5 MySQL database structure.....	17
3.2 Customer interaction modules.	18
3.2.1 General description of modules.....	18
3.2.2 GetScenario - getting data about the scenario.....	19
4. FEEDBACK AND SUPPORT.....	23

1. OVERVIEW OF COMPONENTS

To facilitate orientation, the text of the documentation provides the following color codes for highlighting objects:

Class

Procedure

C# variable

Unity Editor Object

PHP scripts

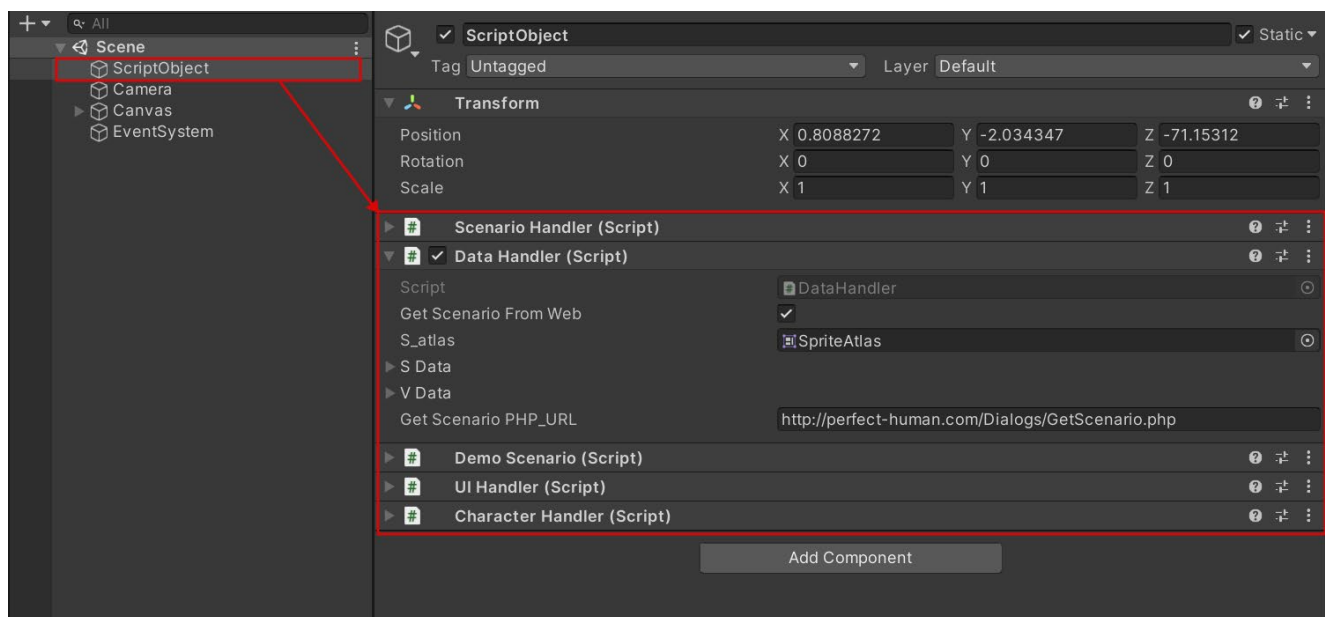
SQL table

PHP variable

Please note that some objects may have multiple color codes. For example, **DataManager.InitSettings()** means that the **InitSettings()** procedure will be called from the **DataManager** singleton class.

1.2 A set of base control classes.

The template uses a set of control C # singleton scripts. Most of these scripts are attached to a single **ScriptObject**.



DataHandler. Sets the data structure, stores information about the received script, and is also responsible for receiving data from the server.

ScenarioHandler. Handler for all scenario actions.

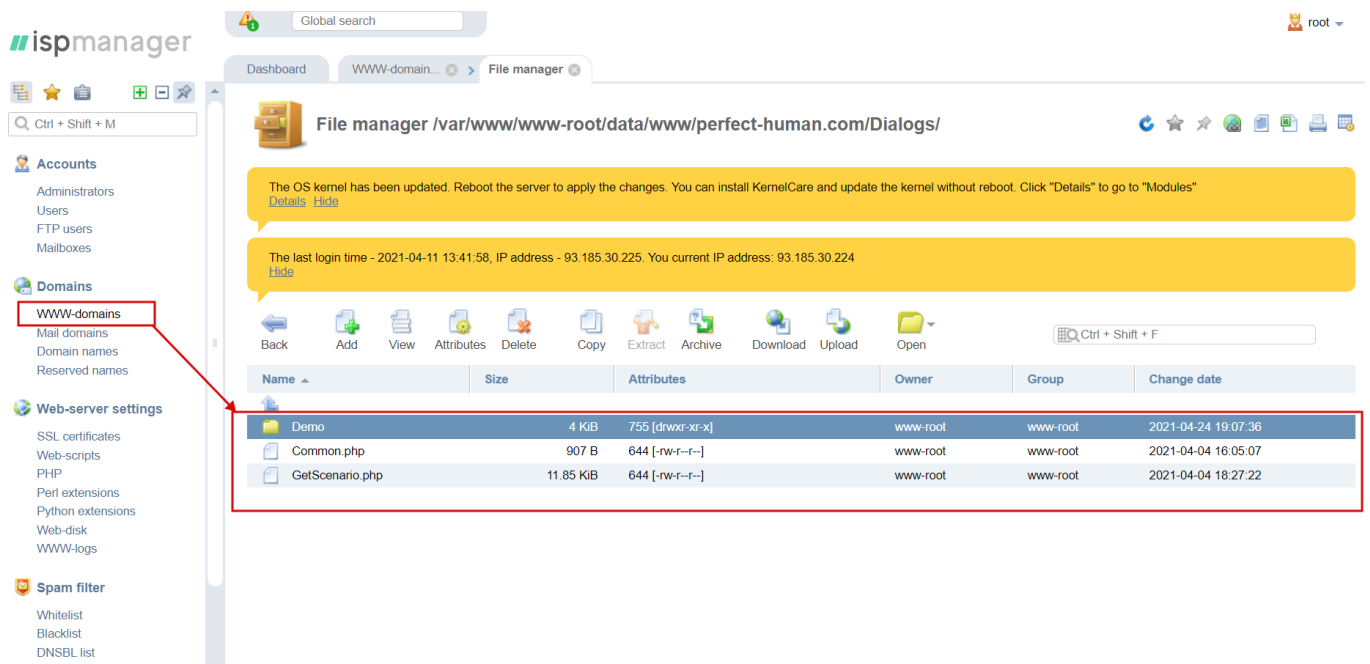
UIHandler. Handler for menu actions.

DemoScenario. Contains a demo script.

StatInfo. Component for displaying character characteristics in the generation list.

SnapScrolling. Horizontal slider controller component (at startup).

1.3 Server-side PHP scripts for interacting with Unity.



The screenshot shows the ISPmanager File manager interface. The main panel displays the directory `/var/www/www-root/data/www/perfect-human.com/Dialogs/`. A red box highlights the **WWW-domains** option in the left sidebar and the file list in the main panel. The file list contains three items:

Name	Size	Attributes	Owner	Group	Change date
Demo	4 KiB	755 [drwxr-xr-x]	www-root	www-root	2021-04-24 19:07:36
Common.php	907 B	644 [-rw-r--r--]	www-root	www-root	2021-04-04 16:05:07
GetScenario.php	11.85 KiB	644 [-rw-r--r--]	www-root	www-root	2021-04-04 18:27:22

These scripts provide communication between the client and the MySQL server. There is no direct connection to the server for security reasons.

GetScenario.php Getting a demo script.

Common.php Generic script, provides service information for connecting to the database and security against SQL injection.

2. CONFIGURING CLIENT MODULES

2.1 Data storage and retrieval module.

2.1.1 Main functions.

All data processing and retrieval is carried out in the **DataHandler** class.

Get_Scenario_from_Web()

Responsible for receiving data from the server. For demonstration purposes, a small script is loaded. After receiving JSON via the API, the result is pushed onto the script class stack with the initialization of all variables. At the end of the procedure, the demo script **ScenarioHandler.StartScenario(sData.scenarios [0])** is called.

GetSpriteFromAtlas(string SpriteID, string Prefix = "")

Searching for sprites in the sprite atlas using **GetSpriteFromAtlas(string SpriteID, string Prefix = "")**, where **SpriteID** is the sprite ID in the atlas, and **Prefix** is an optional prefix if you have many duplicate image names.

AddNewScenarioCharacter(List<ScenarioCharacter> cc, int id, string s_name, int s_place, string s_image = "")

A function for adding a character to a script.

The **id** parameter is the character's identifier.

The **s_name** parameter is a string with the character's name.

The **s_place** parameter is a number, the ordinal number of the character's starting position. 0 - player character. 1 - position to the left. 2 - Right, top. 3 - Right, bottom.

The **s_image** parameter is an identifier string for the character image. It is desirable to place the image in the folder with the atlas.

AddNewStage(List<ScenarioStage> cl, int s_StageOrder, int s_PersonID, string s_StageType = "Talk", int s_Special = 0, float s_Delay = 0)

A function to add a stage to a script.

The **s_StageOrder** parameter is the order of the stage in the script's stage stack.

The **s_PersonID** parameter is the character identifier.

The **s_StageType** parameter is the stage type (1 - check for characteristics, 2 - check for replicas).

The **s_Special** parameter is a pointer to the system stage type with a special handler.

The **s_Delay** parameter is the delay before the new script.

```
AddReplica(List<ScenarioReplica> cr, int s_id, string s_Text, string s_ReplicaType = "Story")
```

A function to add replicas for a specific stage in a script.

The **s_id** parameter is the replica identifier.

The **s_Text** parameter is a string, the text of the replica.

The **s_ReplicaType** parameter is the appearance of the replica cloud.

```
AddNewCondition(List<CommonConditions> cc, int s_IDCheck = 0, int s_TypeOfCondition = 2, string  
s_StatToCheck = "Replica", int s_Min = 1, int s_Max = 0)
```

A function to add conditions for milestones or replicas. It can be used both to determine the conditions of occurrence, and to determine the number of generated character characteristics.

The **s_IDCheck** parameter is the check identifier.

The **s_TypeOfCondition** parameter is the type of check.

The **s_StatToCheck** parameter is the name to be checked.

The **s_Min** parameter is the minimum value to check / add.

The **s_Max** parameter is the maximum value to check / add.

2.2 Scenario scripting module.

2.2.1 Processing the stages of the script.

The start of a stage is performed through the `StartStage(int stageID)` procedure. The only parameter in it is the ordinal identifier of the script stage. This procedure sequentially performs the following actions:

1. Checks the conditions of the stage through `CheckConditions(CurScenarioStage.Conditions)`, if the check fails, the transition to the next stage `NextStage()` is performed.
2. Filling out the list of replicas suitable for this stage. `CheckConditions (curReplica.CheckConditions)` procedure is also applied to them to select matching replicas. If no replicas are found and this is not a special stage, the next stage is performed.
3. If the stage is marked as special, `SystemAction(CurStageType, CurScenarioStage.PersonID)` will be executed.
4. If the stage is normal, then depending on who owns the replica, the animation for the characters is played through `StartNPC()`.
5. If the replica belongs to the player, a set of replicas is created via `MakeVariants()`. If there is only one replica, a regular dialog box with the specified replica type is displayed. If there are several replicas (up to 3), a window of options to choose from is displayed.
6. If the replica belongs to a non-player character, the corresponding replica is displayed via `MakeReplica(CurScenarioStage.PersonID, ReplicaToSend, cr.ReplicaType)`.
7. When the set of stages is exhausted, the demo script is reset to the beginning.

2.2.2 Apply milestone conditions, replicas, and feature generation.

Condition handling is one of the most important operations in script execution. It serves simultaneously for three purposes:

- 1) Selection of stages.
- 2) Selection of replicas.
- 3) Selection of characteristics to generate characteristics of the character. Conditions are of two types:
 1. Checking for previously selected replicas. The condition is considered to be met if the required replica ID was chosen by the player or NPC.
 2. Checking for the current characteristics of the character. The condition is considered fulfilled if the identifier of the `StatToCheck` characteristic of the selected character is included in the range of Min and Max of the condition class.

Both types of selections can be used to create completely arbitrary stage / replica trees, taking into account the context of the narrative.

In addition, the condition class is applied when generating character stats in the `Resolve_Quiz` system stage. The conditions for generation are defined in the `ScenarioReplica` replica class. `SuccessOutcome` on successful check and `ScenarioReplica` replicas. `FailOutcome` if replica verification failed. Characteristics are generated using the `StatToCheck` key.

2.2.3 Handling special events.

Special conditions are handled in the `SystemAction(string StageType, int ReplicaSender)` procedure. It takes the type of the scenario stage as the first parameter; the second stage is needed to define the character to which the event applies. At the time of writing, the following types of stages exist.

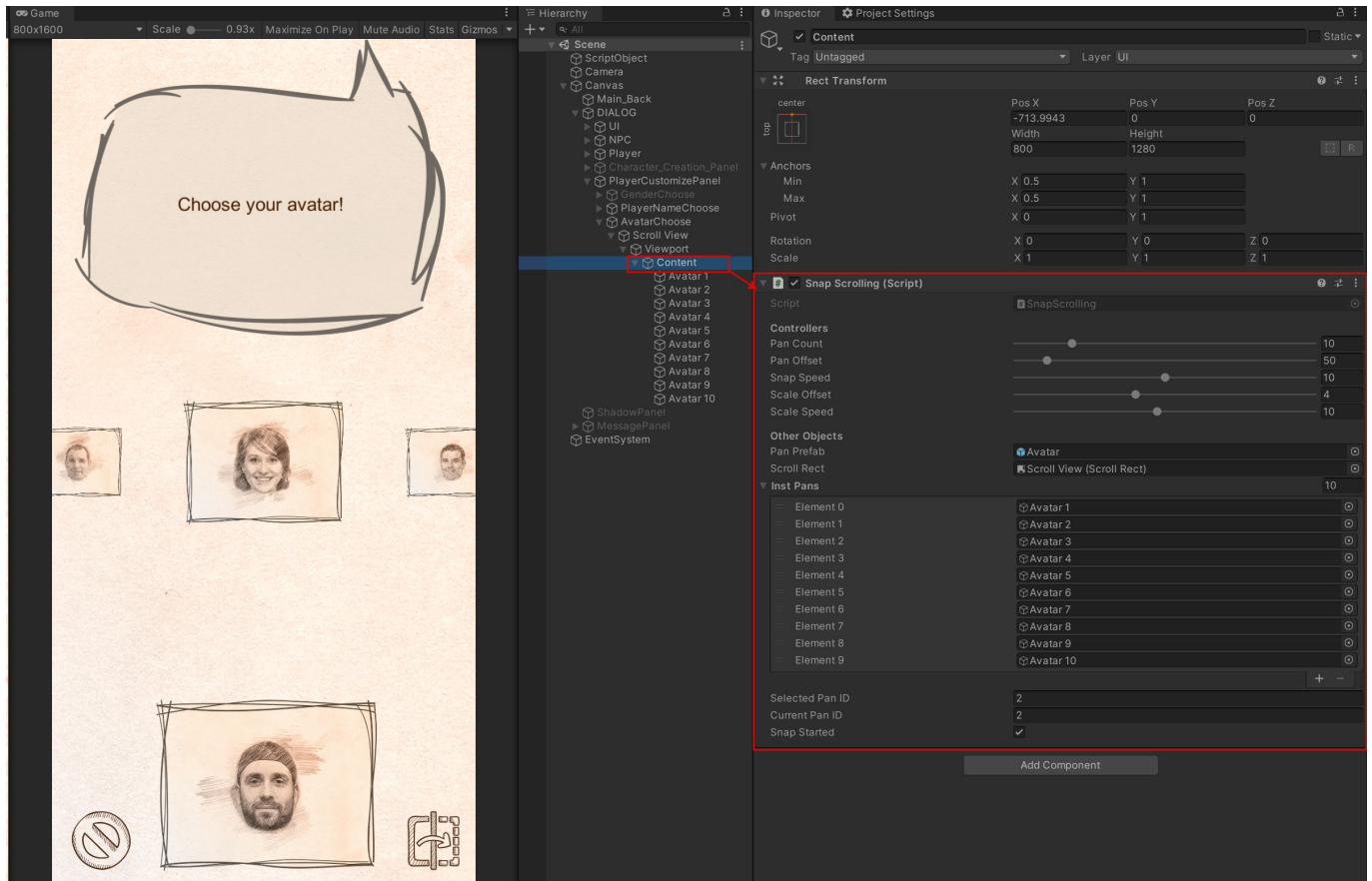
1. **Exit**. This step removes the character from the dialog stack.
2. **Player_Name**. Opens the panel for entering the name of the character.
3. **Player_Avatar**. Opens a panel for selecting a character's avatar.
4. **Player_Gender**. Opens the panel for selecting the character's gender.
5. **Quiz**. Opens a chain of player polling stages.
6. **Resolve_Quiz**. Closes the chain of player-polling stages and opens the panel for generating characteristics earned by the player.
7. **Finish**. Ends the script, regardless of whether there are subsequent steps in the script.

2.2.4 Accrual of characteristics.

Characteristics are accrued in the `EventsActionsResolver()` procedure after confirming the characteristics generation window, called through `SystemAction()` with the **Resolve_Quiz** type. This procedure analyzes all selected stage replicas (**CurScenarioStagesList**) and their associated characteristic arrays in the field of the `ScenarioReplica.SuccessOutcome` class.

2.3 Additional modules.

2.3.1 Avatar selection module.



The avatar is selected when the stage with the **Player_Avatar** type is processed in the **SystemAction()** procedure. The result of the input is processed in the **SystemActionsResolver()** procedure and placed into the **DataHandler.AvatarID** variable, which can be used later. The avatar selection panel is a special carousel-type scrolling object. You can use it as a standalone module in your projects; it allows you to organize your slide views quite flexibly and beautifully. The **SnapScrolling** class is responsible for its work. As pictures, you can either use ready-made slides, or create them programmatically. In the case of manual addition, you need to set the size of the array of slides in the **instPans** variable and fix the elements to it through the Unity inspector. The slider is launched via **StartSnap(int curElem = 0)**. The variable **curElem** points to the array element from which to start the slide show.

You can adjust some aspects of the slider using variables:

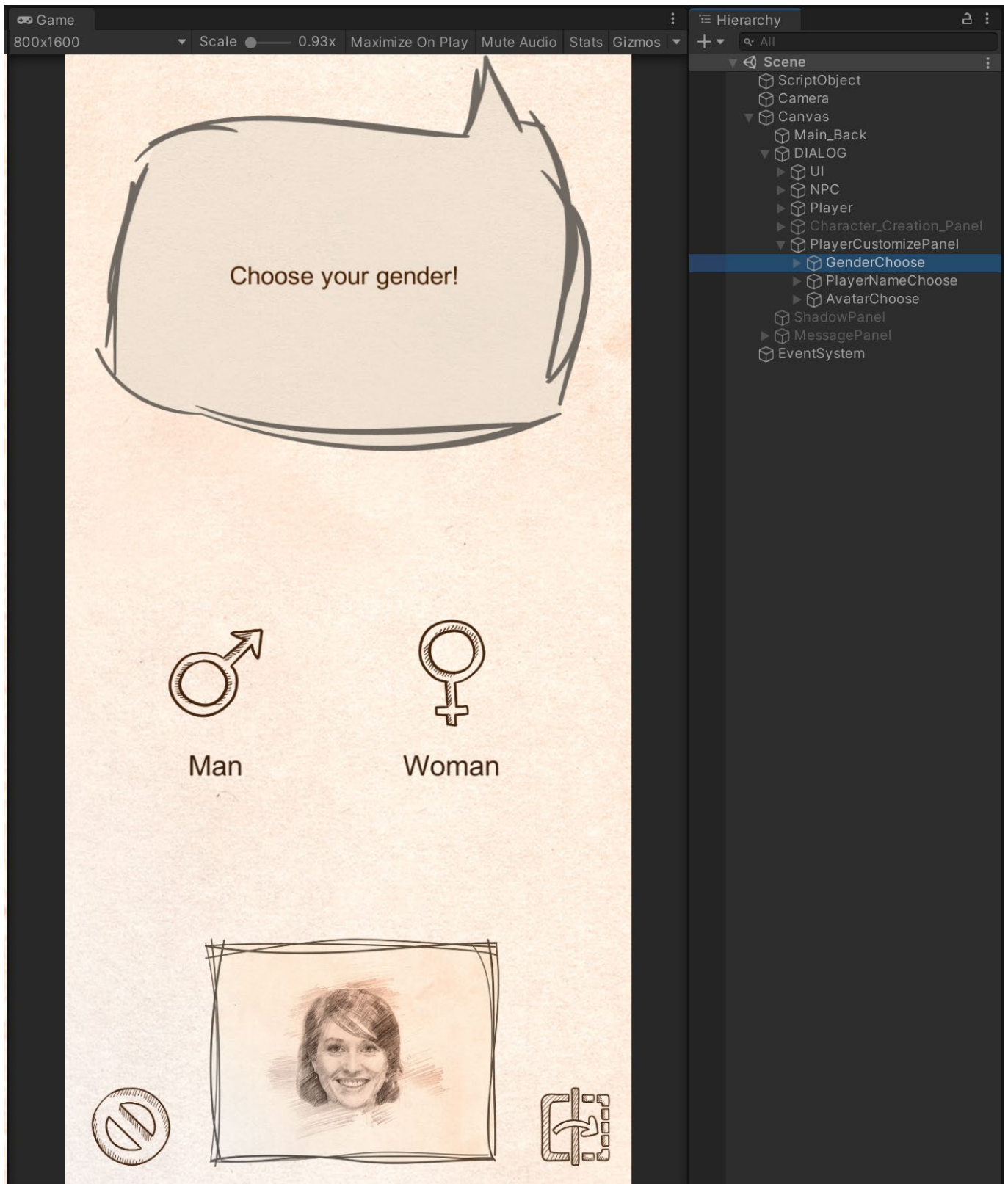
1. **panOffset**. Responsible for the distance between slides.
2. **snapSpeed**. Controls the scrolling speed.
3. **scaleOffset**. Affects the size of the slides.
4. **scaleSpeed**. Adjusts the smoothness of decreasing and enlarging slide elements.

2.3.2 Choosing a name.



The choice of the name occurs when processing a stage with the **Player_Name** type in the **SystemAction()** procedure. The result of the input is processed in the **SystemActionsResolver()** procedure and placed into the **DataHandler.UserName** variable, which can be used later. For this, a special label **[UserName]** can be used in the text of the replica.

2.3.3 Gender selection.



The choice of the name occurs when processing a stage with the `Player_Gender` type in the `SystemAction()` procedure. The result of the input is processed in the `GenderChoose(int gender)` procedure and placed into the `DataHandler.Gender` variable, which can be used later.

2.4 Programmatic scenario input.

2.4.1 Description of the demo script.

The demo script is designed to illustrate the basic functionality of the template. It implements a decision tree, a reaction to the choice of this tree later, a questionnaire with the generation of character characteristics, as well as auxiliary functions such as choosing an avatar, name and gender. This list of system functions can be easily expanded into the original functionality. The programmatic input of this scenario is implemented in the **DemoScenario** class.

All elements of the script are entered using static functions in the **DataHandler** class. These are detailed in section 2.1.1. The finished script is placed in the **DataHandler.sData** variable. It contains a list of all scripts received both programmatically and from the server.

Next, we will dwell in detail on the basic elements of program input.

2.4.2 Creation of script characters.

Character creation is performed using the **DataHandler.AddNewScenarioCharacter()** static function. Add 2 to 4 characters to the scenario. In addition, the zero value for the Place parameter is the player's character. The rest of the sequence numbers are responsible for the order in which they are displayed during startup initialization.

Alternatively, you can assign the required characteristics to each of them using **player.CharValues.Add(new DataHandler.ScenarioCharacterValues(int ValueID_s = 0, int amount_s = 0));**

In this function, the **ValueID_s** parameter is the characteristic identifier defined in the database, or entered manually in the **DemoScenario.AddDefaultValueData()** procedure.

2.4.3 Entering the stages of the script.

After defining the characters, we proceed to create the stages of the script. This is done using the **DataHandler.AddNewStage (IntroScenario.Stages, 1, 2)** function, which returns a reference to the generated script step. The first parameter of this function is a list of stages in the script, the second is the ordinal number of the stage, and the third is the character of the stage.

When entering stages, it is necessary to imagine the plot of the scenario in advance. First of all, you need to understand whether you will have system stages in it that require additional processing, or whether you will have a continuous dialogue with fixing the consequences of your choice.

When chaining the steps, keep in mind that any of the steps can be skipped if the step conditions are not met if they are specified. For example, if in the previous stage you chose the first line, and not the second, the next stage (the character's exit from the stage) can be skipped. This allows the maximum flexibility to implement not only replicas, but also all interactive actions.

2.4.4 Adding replicas to stages.

Replicas are the basic building block of a conversational system. Set using the `DataHandler.AddReplica` function(`s3.ReplicaList`, `4`, "Okay, then let's get started!", "Action");

The first parameter is a list of replicas of the stage to which it will be added. This list can contain as many items as you want. However, only three or fewer of them will be withdrawn. Accordingly, the rest should be filtered using conditions - either for remarks or for character characteristics (For more details, see section 2.2.2). The second parameter is the replica ID. The third is the replica text. The fourth is the appearance of the replica.

Please note that special insertions such as "Good choice, [UserName]!" Are allowed in the text of the replicas. They serve for post processing of text and replacing them with keywords. In this example, replacing it with the character's nickname.

When adding replicas, remember that they are the engine of the whole story. The conditions for subsequent stages, replicas, accrual of characteristics or resources will depend on their choice.

3. CONFIGURING SERVER MODULES

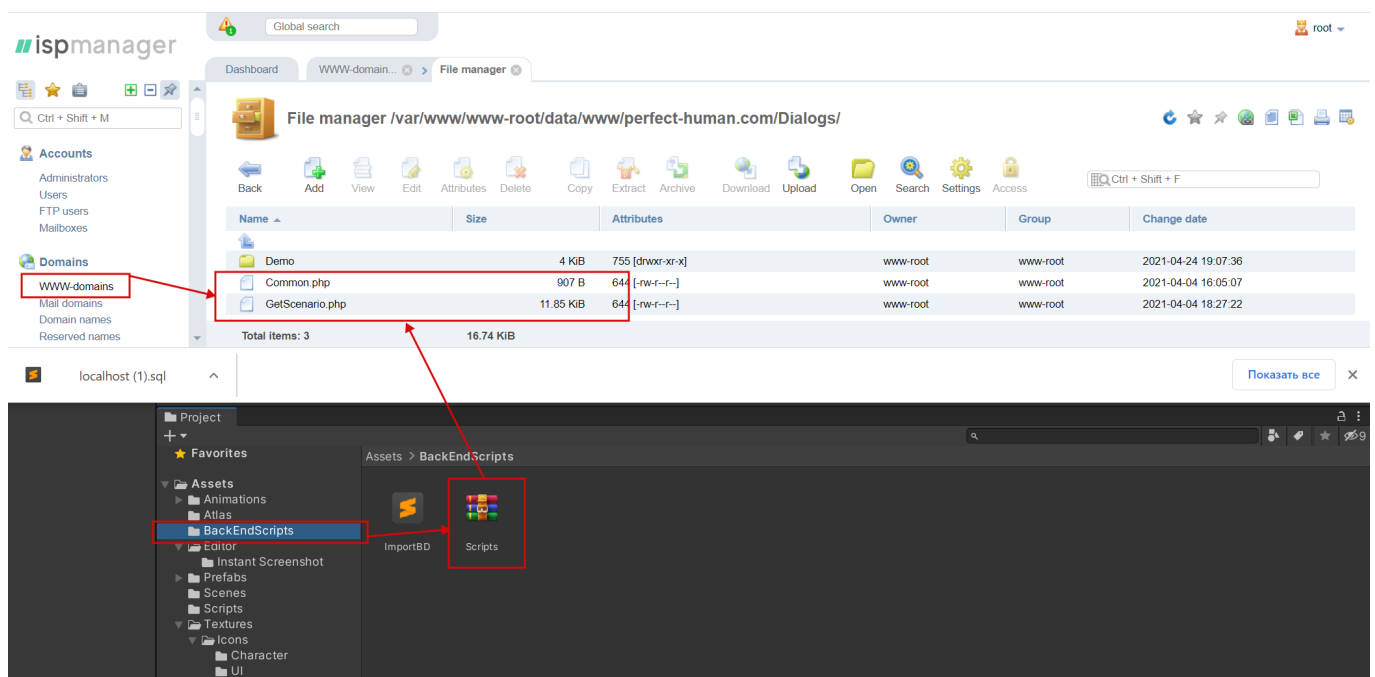
3.1 General information for working with hosting.

3.1.1 Buying hosting and getting data for work.

You can host the back end server either locally on your machine (using Apache or Nginx) or using hosting services. The first option will require you to provide constant access to broadband network and network administration, it is more preferable when your project is sufficiently developed or at the testing stage. However, to begin with, the hosting option is much more preferable. We will consider it.

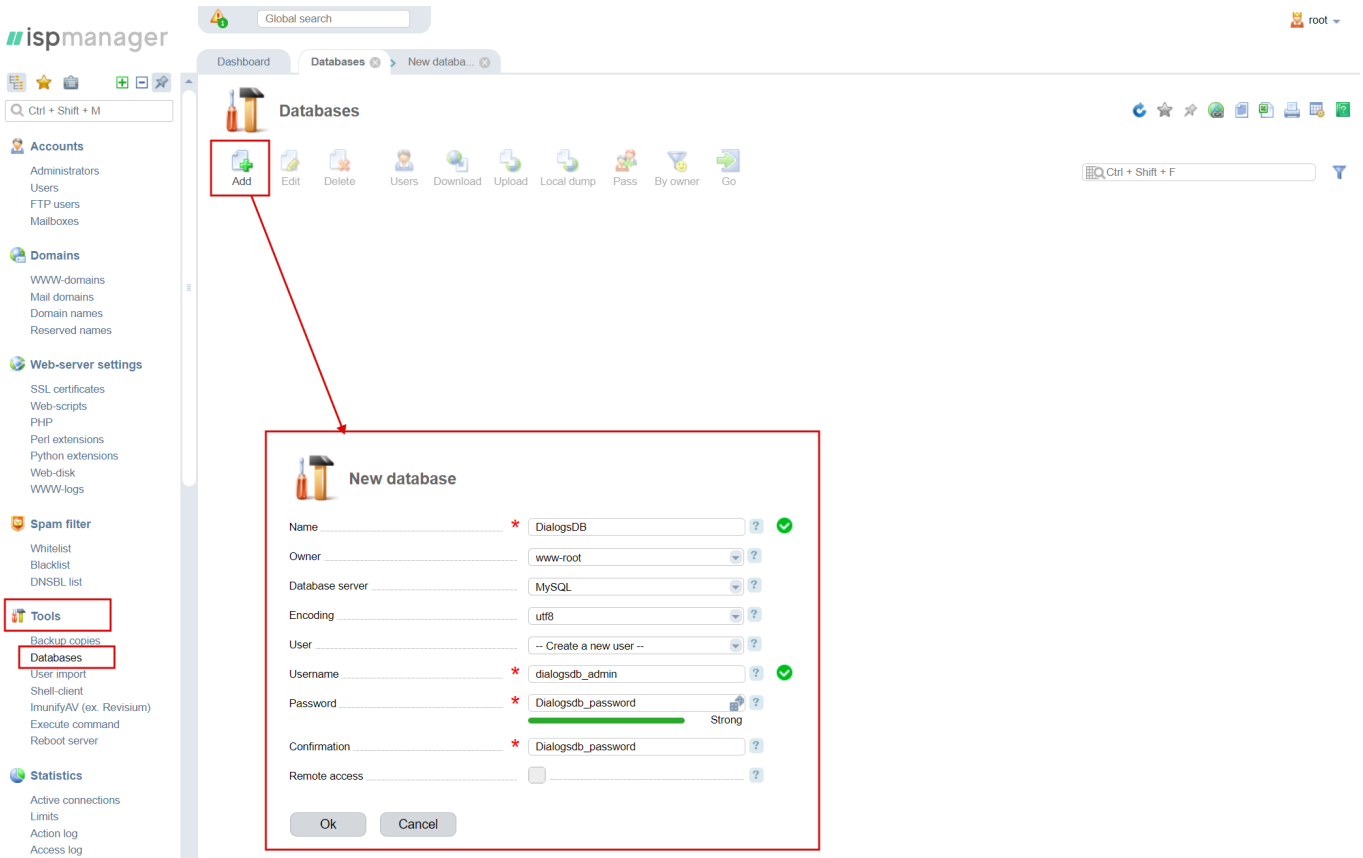
To get started, you need to purchase hosting space. These services are provided by many companies, choose any that suits you best. To do this, just type “hosting services” in your browser. In this documentation, all examples are dealt with in ISP panel for file administration and PhpMyAdmin for database administration. Therefore, if it is important for you to repeat the process of deploying files and databases systematically, make sure that you have them installed. If you already have hosting and skills to work with it, skip this section.

3.1.2 Writing script files to the server through ISP panel.

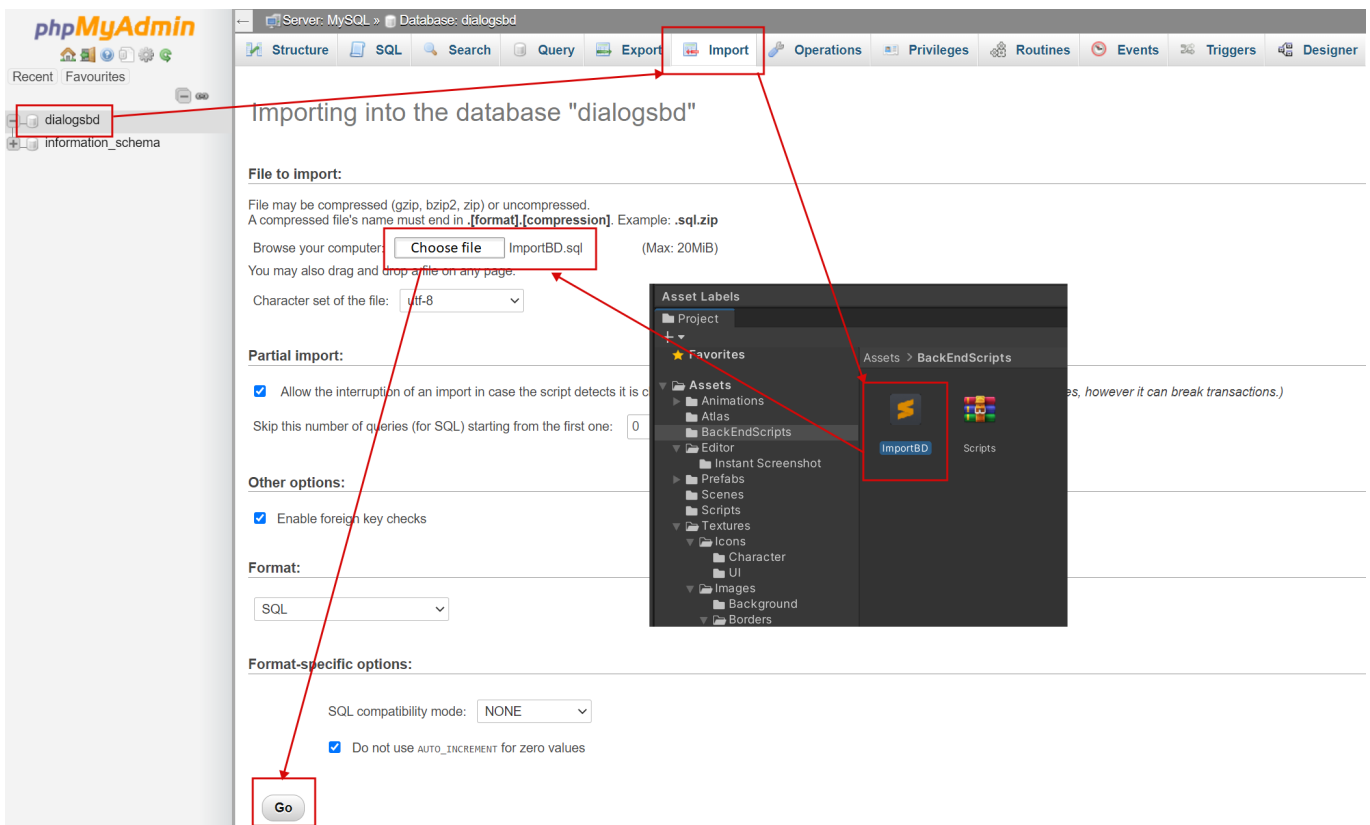


The archive with all scripts for the server is stored in the `BackEndScripts` directory. Download it and place it in the space that suits you. Unzip the content.

3.1.3 Importing database into phpMyAdmin.



Create a new database, user and password to it. Be sure to remember, or better write down this data somewhere. We need them to write in the database connection script.



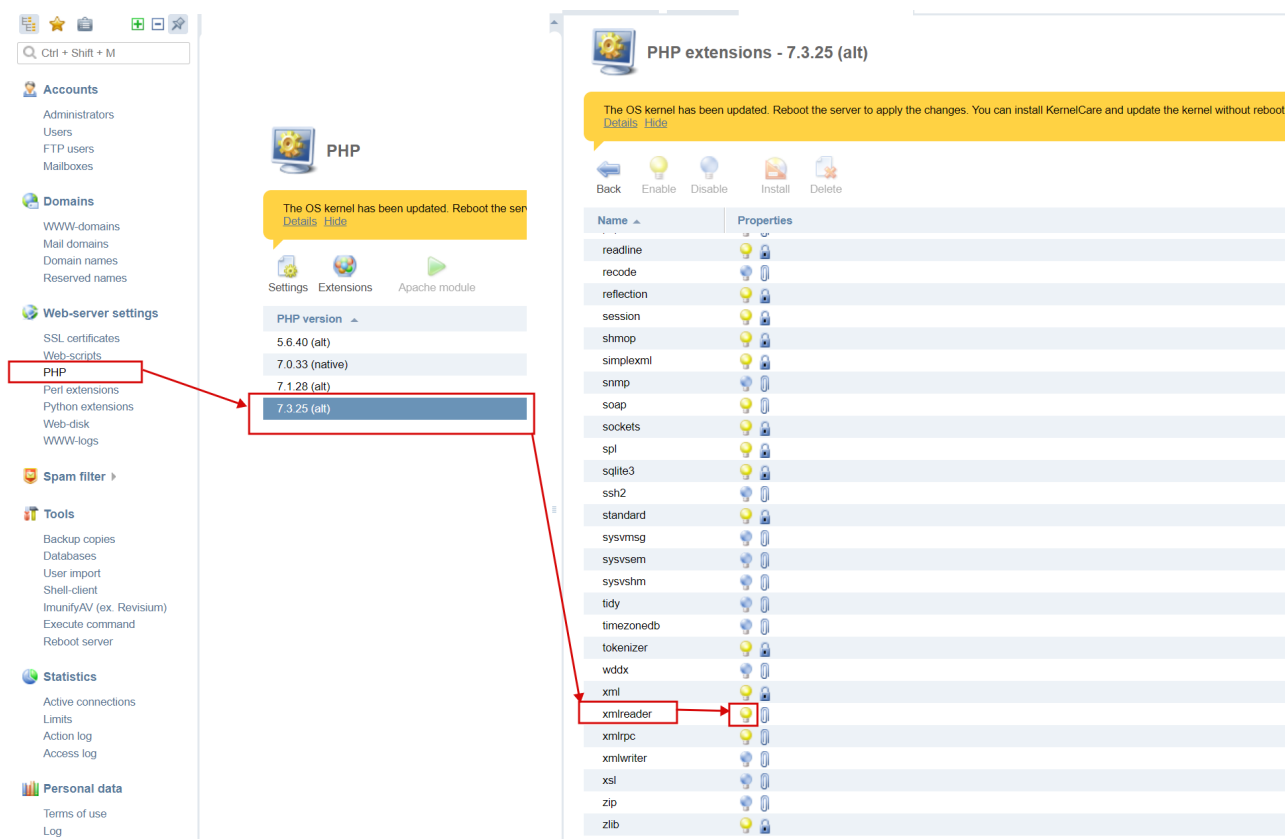
Import the file with the Database structure and initial settings. It does not yet contain the replicas themselves, we will upload them later. When the import is complete, you will see a screen like this:



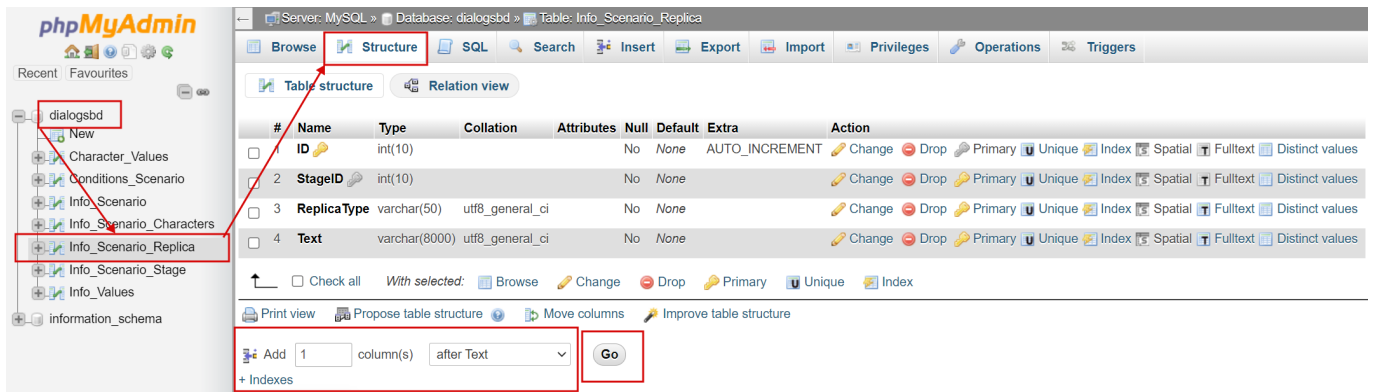
Congratulations! Database loaded successfully. I recommend that you study the structure of the database well in order to understand all the features of data storage. Once you understand how structure correlates with structure within the application, you can begin to tailor the structure to your needs.

3.1.4 PHP version and required libraries.

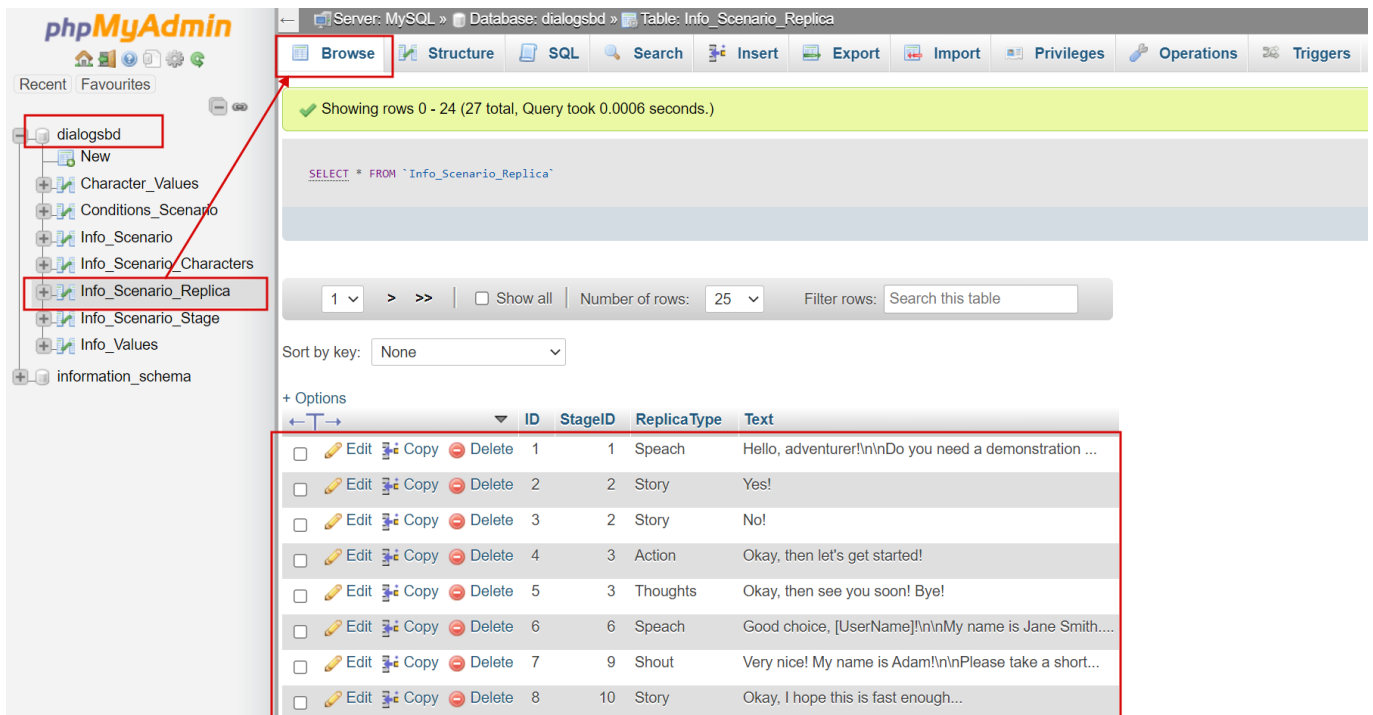
Important! Be sure to pay attention to your PHP version. If you have version 7+, no changes are required. However, if you have an older version of PHP, use the scripts from the folder (PHP 5+) in **BackEndScripts**. This is important because the calls to the SQL server are different in these versions.



3.1.5 MySQL database structure.



The structure of the Database is determined by a set of tables and columns in these tables. The example above shows the structure of a table with replicas. It has 4 columns. The replicas themselves are represented in it by rows, where each value of the column reflects some characteristic. You can view them in the Browse menu.



Note that the columns are exactly the same structure as the **DataHandler.ScenarioReplica** class in Unity. This is not accidental; in general, data homogeneity should be maintained both on the server and on the client. Try to avoid unnecessary fields (columns) in your structure. To add a new column, you can use the menu below, define the number of columns and the starting position in it and click Go. To edit the columns, use the Change button opposite each column.

Try not to change the structure unnecessarily, because without a clear understanding, there is a possibility of a mismatch between your scripts and the structure of the database, which will lead to errors when loading the application or importing data. Always change the structure consciously. If you

have irreparably damaged the database, try to restore it, for this use, more often use table backups. To do this, use the Export option.

Now let's take a look at what each table is used for.

1. **Info_Scenario**. This table is used to store all scenarios. Here you set the name and identifier of the script. Duplicates the structure of the **DataHandler.ScenarioData** class.
2. **Info_Scenario_Stage**. Table with the stages of the scenario. Contains a link to the script ID. Duplicates the structure of the **DataHandler.ScenarioStage** class.
3. **Info_Scenario_Replica**. Table with replicas of the stages of the scenario. Contains a link to the replica id. Duplicates the structure of the **DataHandler.ScenarioReplica** class.
4. **Conditions_Scenario**. A table with a full description of the conditions for stages and replicas. Duplicates the structure of the **DataHandler.CommonConditions** class.
5. **Info_Scenario_Characters**. This table serves as a repository for all characters linked through a scenario ID. Duplicates the structure of the **DataHandler.ScenarioCharacter** class.
6. **Character_Values**. Table with a set of character characteristics. Contains a link to the id of the characters. Duplicates the structure of the **DataHandler.ScenarioCharacterValues** class.
7. **Info_Values**. A reference table with a description of all characteristics and / or resources. Duplicates the structure of the **DataHandler.ValueData** class.

3.2 Customer interaction modules.

3.2.1 General description of modules.

Let's take a closer look at the PHP scripts that we placed earlier on our server. They provide client-server communication through the **DataHandler** class.

Unity allows you to work directly with a remote MySQL server. To do this, just open a port on the server and specify the connection data on the Unity side. From a performance point of view, this option is even more optimal, since it removes the overhead of serializing and deserializing JSON. However, from a security point of view, opening free access to the database is extremely dangerous. Experience suggests that the only correct solution is access through the localhost of the server itself. That is why we need a special transport for sending Post and Get requests to our SQL server. In general, the structure of these php scripts is similar:

1. The parameters `$_POST['YOUR_VARIABLE']` are accepted as input.
2. The security key is verified using MD5 encryption. This key is specified in the **Common.php** (variable `$secretKey`).
3. A connection to the `dbConnect()` database is opened. Connection parameters and the function itself are described in the **Common.php** script)

4. The database is being read or modified.
5. The response of the MySQL server is processed as an array.
6. The response is encoded in JSON format using the `json_encode()` directive
7. Printing the result through the `echo` method. If successful, the `SUCCESS` keyword is usually added, which serves as a criterion to start parsing the result on the client.
8. Closing the connection to the database via `mysql_close($link);`

Important! Before executing any queries, make sure that you have filled in all the variables in the given script correctly.

1. The `$secretKey` variable must exactly match the `DataHandler` variable. `SecretKey` in Unity Editor.
2. The `$dbName` variable must match the name of the database you created in section 3.1.3.
3. In the directive `$link = mysql_connect('localhost', $dbName, 'YOUR_PASSWORD');` replace the password set in section 3.1.3
4. Make sure that all paths in the `DataHandler` class match the full path to each of your scripts. By default, there are paths to the Perfect-Human server with demo data.

3.2.2 GetScenario - getting data about the scenario.

The `GetScenario.php` script is the most important because it is responsible for getting scenario data. We will move through the functional blocks starting from top to bottom.

`include("Common.php");` This string connects the shared module, which stores the database connection string and secret key.

Parameters are set in WWWForm on the client side. On the server side, they are accepted like this:

`$hash = safe($_POST ['hash']);` This is a special hash for encrypting the secret key. Typically, it is used in conjunction with a user ID. You can add your token to this field.

The following construct verifies the hash on the server side and on the client side. If they do not match (someone knocked on the server from the outside), a return from the script with the message "Access Denied!" is triggered.

```
$real_hash = md5($secretKey);
```

```
if ($real_hash != $hash)
```

```
{
    die("Access Denied!");
}
```

Request to obtain a reference dictionary of characteristics.

```
//-----  
//echo("VALUES INFO");  
//-----  
  
$query = "SELECT * FROM `Info_Values` AS Info_Values";  
$Info_Values_Result = mysqli_query($link, $Info_Values_query) or die(mysqli_error());  
  
$Val_response = array();  
$Val_data = array();  
  
while($row_val = mysqli_fetch_assoc($Info_Values_Result))  
{  
    $Val_ValueID = $row_val['ValueID'];  
    $Val_Name = $row_val['Name'];  
    $Val_ImageID = $row_val['ImageID'];  
  
    $Val_data[] = array('ValueID'=> $Val_ValueID, 'Name'=> $Val_Name, 'ImageID'=> $Val_ImageID);  
}  
  
$Val_response['values'] = $Val_data;  
$Val_json = json_encode($Val_response);  
  
echo $Val_json;  
echo "|";
```

What you should pay attention to here:

1. SELECT with the * sign means that we are interested in all the fields from the `Info_Values` table.
2. The FROM keyword indicates the main data source for the query, in our case it is the table ``Info_Values` AS Info_Values`. Pay attention to the alias of this table `Info_Values`, this name can be arbitrary, but meaningful enough to make the query easier to read.
3. After the request, an array with a response is assembled to assemble a response in JSON format using `json_encode($Val_response)`;

This is followed by 2 requests to get characters and their characteristics.

```
//-----  
//echo("SCENARIO CHARACTERS");  
//-----  
  
$CharacterScenQuery = "  
  
SELECT  
Characters.ID ID  
,  
Characters.ScenarioID ScenarioID  
,  
Characters.Place Place  
,  
Characters.Name Name  
,  
Characters.ImageID ImageID  
  
FROM  
`Info_Scenario_Characters` AS Characters";  
  
$CharactersScenResult = mysqli_query($link, $CharacterScenQuery) or die(mysqli_error());  
  
//-----  
//echo("FILLING CHARACTERS");  
//-----  
  
while($rowch = mysqli_fetch_assoc($CharactersScenResult))  
{  
    $charid = $rowch['ID'];
```

```

$character_stats = array();

while($rowstat = mysqli_fetch_assoc($CharactersStatsResult))
{
    if($charid == $rowstat['CharacterID'])
    {
        $character_stats[] = array('ValueID'=> $rowstat['ValueID'], 'Amount'=> $rowstat['Amount']);
    }
}

$character_data[] = array('ID'=> $charid, 'ScenarioID' => $rowch['ScenarioID'], 'Name'=> $rowch['Name'], 'Place'=> $rowch['Place'], 'ImageID'=>
$rowch['ImageID'], 'CharValues' => $character_stats);
}

$character_response['characters'] = $character_data;
$character_json = json_encode($character_response);

echo $character_json;
echo "|";

```

Note that when assembling the last item, a nested loop through the character stats array is used.

```

//-----
//echo("SCENARIOS");
//-----

$$ScenarioQuery = "
SELECT
Info_Scenario.ID ScenarioID
,
Info_Scenario.Name Name
,
Conditions_Scenario.ID Conditions_ID
,
Conditions_Scenario.TypeOfCondition TypeOfCondition
,
Conditions_Scenario.StatToCheck StatToCheck
,
Conditions_Scenario.IDCheck IDCheck
,
Conditions_Scenario.Min Min
,
Conditions_Scenario.Max Max
FROM
`Info_Scenario` AS Info_Scenario
LEFT JOIN
Conditions_Scenario AS Conditions_Scenario ON Info_Scenario.ID = Conditions_Scenario.LinkID AND Conditions_Scenario.TypeOfLink = 1";

$scenarioResult = mysqli_query($link, $ScenarioQuery) or die(mysqli_error());

```

What you should pay attention to here:

1. SELECT and the fields behind it tell us which table fields we are interested in. Please note that the data is taken from different tables. For example, **Conditions_Scenario.ID**. Here, the condition identifier is taken from the **Conditions_Scenario** table. We will consider the rules for joining two or more tables in paragraph 3.
2. The LEFT and INNER JOIN constructs are required to join data to the main selection. For example, our character stats table has stats IDs. But there is no information about the characteristics themselves (names and properties). To make them available in the SELECT selection, it is necessary to join two tables in which the characteristic identifier is common.
3. Thus, the construction: LEFT JOIN **Conditions_Scenario** ON **Info_Scenario.ID = Conditions_Scenario.LinkID AND Conditions_Scenario.TypeOfLink = 1** allows us to join the scenario ID from the **Info_Scenario** table and the conditions ID from the **Conditions_Scenario** table.
4. Differences between LEFT and INNER, roughly speaking, are as follows: LEFT or left join means that

you can select data from the table that you want to join (**Conditions_Scenario**) to the main table (**Info_Scenario**). INNER or inner join means that the records of the main table must necessarily match the **Conditions_Scenario** table according to the rule of joining them. If they do not match at all, an empty response to the request will be returned to you. Thus, under certain conditions, INNER JOIN can be used as a filter.

In addition, here we also see the condition **TypeOfLink** = 1. This condition selects only conditions that match the type. This is a VERY important point. Identifiers can overlap within Scenarios, Stages and Replicas:

TypeOfLink = 1 - connection by script identifiers.

TypeOfLink = 2 - connection by Stage identifiers.

TypeOfLink = 3 - connection by Replica identifiers.

TypeOfLink = 4 - connection by identifiers of successful Replica outcomes.

TypeOfLink = 5 - connection by identifiers of unsuccessful Replica outcomes.

The next two requests - to Stages and Replicas are executed according to the same scheme. Note that we are returning multiple arrays with vertical bar-separated data. This is a separator symbol. On the client, we will process this result in the **DataHandler.Get_Scenario_Web_Process()** procedure:

```
// Separation of two data sets.  
string[] mSplit = result.Split("|"[0]);  
  
// Converting JSON from the server response directly to the DataManager.ValuesData class.  
vData = JsonUtility.FromJson<ValuesData>(mSplit[1]);
```

At the very end of the script, you need to close the MySQL connection.

```
mysqli_close($link);
```

4. FEEDBACK AND SUPPORT.

Thank you for purchasing and using the Sketch Dialogs template.

I hope you enjoy your purchase and its implementation will not cause much difficulty. If you have any questions, please email me PerfectHumanApps@gmail.com or the Sketch Dialogs support section of the Unity Forum. This will help not only you, but also myself.

Also, I would appreciate it if you could take the time to provide feedback on the Asset Store page!