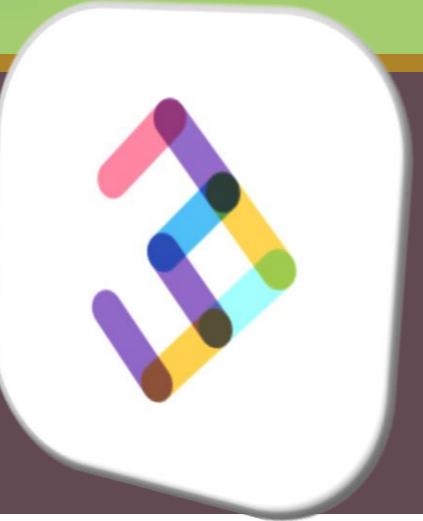




Design and Implementation of Online Experiments

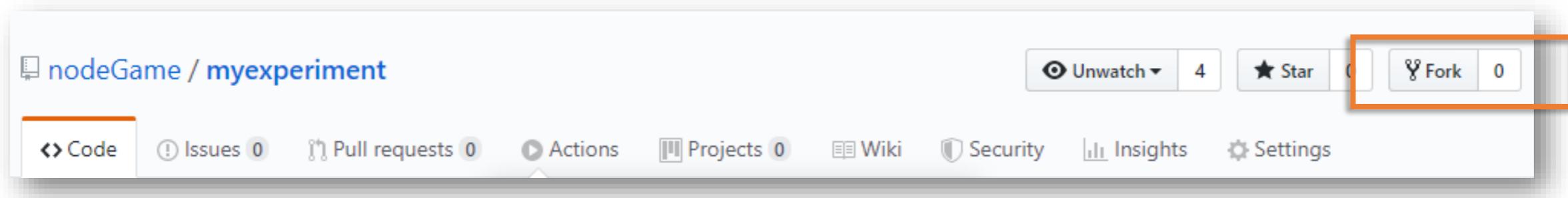


nodeGame.org
Stefano Balietti
MZES and Heidelberg
nodeGame
Advanced

@balietti
@nodegameorg
stefanobalietti.com@gmail.com

Clone My Experiment

1. Do you already have a GitHub account? Get one!
2. Go to <https://github.com/nodeGame/myexperiment>
3. Fork the repository to get a copy under your account



Confused about forking? Learn about forking and "spooning" in software programming.

<https://www.youtube.com/watch?v=8wUOUmeulNs>

Clone My Experiment

4. Copy the link to clone your **forked** repository

The screenshot shows a GitHub repository page for `shakty / myexperiment`, which is a fork of `nodeGame/myexperiment`. The repository has 4 commits, 1 branch, 0 packages, 0 releases, 1 contributor, and is licensed under MIT.

The main navigation bar includes links for Code, Pull requests (0), Actions, Projects (0), Wiki, Security, Insights, and Settings. Below the navigation bar, there is a brief description: "A Sample Game to Learn Features of the NodeGame Framework".

On the right side of the page, there is a "Clone or download" button, which is highlighted with a red box. To the right of this button is a "Copy to clipboard" icon, also highlighted with a red box. A yellow arrow points from the text "Copy to clipboard" to this icon.

The repository listing shows several files and folders:

- `shakty README`
- `auth` - My First Commit
- `channel` - My First Commit
- `game` - Game with Hands On Exercises Comments

At the bottom of the page, there are links for "Open in Desktop" and "Download ZIP". A timestamp at the bottom right indicates the page was last updated 10 minutes ago.

Clone My Experiment

5. Locate the games/ folder inside the nodeGame folder
6. Clone the forked repository to your local hard drive

```
balistef@mzes072 MINGW64 ~/Desktop/nodegame-v5.4.0-dev/games
$ git clone
      Open
      Copy   Ctrl+Ins
      Paste  Shift+Ins
      Select All
      Search  Alt+F3
      Reset   Alt+F8
      Default Size Alt+F10
      Scrollbar
      Full Screen Alt+F11
      Flip Screen Alt+F12
      Options...
      ✓
balistef@mzes072 MINGW64 ~/Desktop/nodegame-v5.4.0-dev/games
$ git clone git@github.com:shakty/myexperiment.git
Cloning into 'myexperiment'...
remote: Enumerating objects: 62, done.
remote: Counting objects: 100% (62/62), done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 62 (delta 13), reused 62 (delta 13), pack-reused 0
Receiving objects: 100% (62/62), 29.71 KiB | 338.00 KiB/s, done.
Resolving deltas: 100% (13/13), done.

balistef@mzes072 MINGW64 ~/Desktop/nodegame-v5.4.0-dev/games
$ |
```

Link will be slightly different for you

Clone My Experiment

7. Open project in ATOM and try:
Packages/Bracket Matcher
Edit/Lines/Auto Indent
Publish Changes on GitHub
Extra packages: JSHINT, Teletype, etc.

The screenshot shows the Atom code editor interface. On the left, the 'Project' sidebar displays a directory structure for a project named 'myexperiment'. The 'player.js' file is selected and shown in the main editor area. The code in player.js is as follows:

```
/*
 * # Player type implementation of the game stages
 * Copyright) 2019 Stefano <Info@nodegame.org>
 * MIT Licensed
 *
 * Each client type must extend / implement the stages defined in 'game.stages'.
 * Upon connection each client is assigned a client type and it is automatically
 * setup with it.
 *
 * http://www.nodegame.org
 * ...
 */
'use strict';

module.exports = function(treatmentName, settings, stager, setup, gameRoom) {
  stager.setOnInit(function() {
    // Initialize the client.
    var header, frame;

    // Setup page: header + frame.
    header = W.generateHeader();
    frame = W.generateFrame();

    // Add widgets.
    // this = node.game.
    this.visualRound = node.widgets.append('visualRound', header);
    this.visualTimer = node.widgets.append('visualTimer', header);
    this.doneButton = node.widgets.append('DoneButton', header);

    // Bid is valid if it is a number between 0 and 100.
    this.isValidBid = function(n) {
      return node.JSUS.isInt(n, -1, 101);
    };

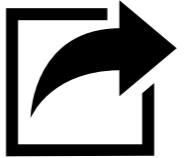
    // Additional debug information while developing the game.
    // this.debugInfo = node.widgets.append('DebugInfo', header)
  });
}
```

Below the editor, the 'Linter' panel shows two JSHint warnings:

Severity	Provider	Description	Line
Warning	JSHint	W097-Use the function form of 'use strict'.	14:1
Warning	JSHint	W117-'module' is not defined.	16:1

On the right side of the screen, there are GitHub integration panels for 'Unstaged Changes' (No changes), 'Staged Changes' (No changes), and a 'Commit message' field with a 'Commit to master' button. The status bar at the bottom indicates 'CRLF' and 'UTF-8' encoding, and shows the current branch is 'master'.

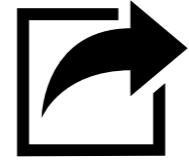
Exchanging Messages



[Data-Exchange-Examples](#)

Command	Description	Recipient code required
node.done	terminates current step, and sends any optional data to server	no, any data is automatically inserted in database: node.game.memory
node.say	sends data to other clients or server	yes, must implement event listener: node.on.data

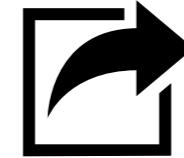
Exchanging Messages



[Data-Exchange-Examples](#)

Command	Description	Recipient code required
node.done	terminates current step, and sends any optional data to server	no, any data is automatically inserted in database: node.game.memory
node.say	sends data to other clients or server	yes, must implement event listener: node.on.data
node.set	sends data to database	no, any data is automatically inserted in database: node.game.memory
node.get	invokes a remote function, and waits for return value	yes, must implement event listener: node.on

Examples: node.done



[Step-Callback-Functions-v5](#)

Dictator



I say 99 for me
and 1 for you.

```
// Get offer from input
// onclick send to server.
var offer = W.gid("offer").value; // 1
var submit = W.getElementById("submitOffer");

submit.onclick = function() {
  node.done({ offer: offer });
};

node.done terminates current step and send
any input parameter to the server
```

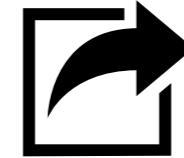
Observer



Waiting for a game message labeled "decision"
from *someone* (server or player), then ends step.

```
node.on.data("decision", function(msg) {
  // msg.data = 1;
  var offer = "The offer is: " + msg.data;
  // Display offer using W.
  W.setInnerHTML("decision", offer);
  node.done();
});
```

Examples: node.done



[Step-Callback-Functions-v5](#)

Dictator



I say 99 for me
and 1 for you.

```
// Get offer from input
// onclick send to server.
var offer = W.gid("offer").value; // 1
var submit = W.getElementById("submitOffer");

submit.onclick = function() {
  node.done({ offer: offer });
};
```

When both players
are **DONE** the
game advances to
next step.

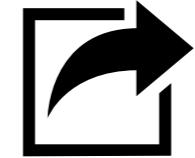
Observer



[player.js](#)

```
node.on.data("decision", function(msg) {
  // msg.data = 1;
  var offer = "The offer is: " + msg.data;
  // Display offer using W.
  W.setInnerHTML("decision", offer);
  node.done();
});
```

Logic and Player Exchanging Messages



[Step-Callback-Functions-v5](#)

Dictator

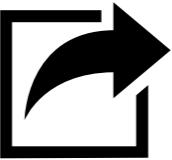


```
node.done({ offer: offer });
```

Sends info to server database and ends step



Logic and Player Exchanging Messages



[Step-Callback-Functions-v5](#)

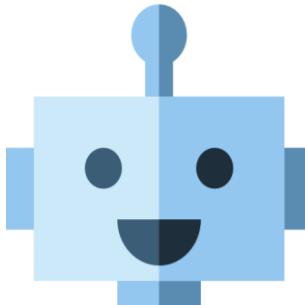
Dictator



```
node.done({ offer: offer });
```

Sends info to server database and ends step

Logic



```
node.on.data('done', function(msg) {  
    var observer = node.game.matcher.getMatchFor(msg.from);  
    // Send the decision to the other player.  
    node.say('decision', observer, msg.data.offer);  
});
```

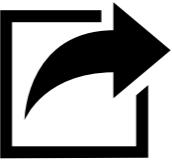


Icon by Freepik

Receives the input (could do additional computation), gets the observer ID with matcher API, and forwards the offer with node.say



Logic and Player Exchanging Messages



[Step-Callback-Functions-v5](#)

Dictator

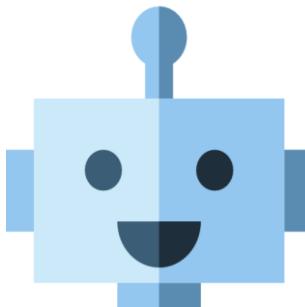


```
node.done({ offer: offer });
```

Sends info to server database and ends step



Logic



```
node.on.data('done', function(msg) {  
    var observer = node.game.matcher.getMatchFor(msg.from);  
    // Send the decision to the other player.  
    node.say('decision', observer, msg.data.offer);  
});
```



Receives the input (could do additional computation), gets the observer ID with matcher API, and forwards the offer with node.say

Observer



```
node.on.data("decision", function(msg) {  
    // Display offer.  
    node.done();  
});
```

Displays the offer and ends the step.



Icon by Freepik

Understanding node.say

```
node.say('decision', observer, msg.data.offer);
```

node.say takes 3 parameters, the last two of which are optional.



Understanding node.say

```
node.say('decision', observer, msg.data.offer);
```

node.say takes 3 parameters, the last two of which are optional.

1. The label is a string summarizing the purpose of the message

Understanding node.say

```
node.say('decision', observer, msg.data.offer);
```

node.say takes 3 parameters, the last two of which are optional.

1. The label is a string summarizing the **purpose of the message**.
2. The recipient is a string containing the **ID of the recipient** of the message (optional, if missing message is sent to the logic).

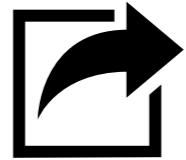
Understanding node.say

```
node.say('decision', observer, msg.data.offer);
```

node.say takes 3 parameters, the last two of which are optional.

1. The label is a string summarizing the **purpose of the message**.
2. The recipient is a string containing the **ID of the recipient** of the message (optional, if missing message is sent to the logic).
3. A **payload**, that is additional data attached to the message (optional). Here, it is the value of the offer.

Sending and Receiving Game Messages



[Game-Messages-v5](#)

When sending a message with any of the interactive methods (say, set, get, done), an object of class Game Message is created

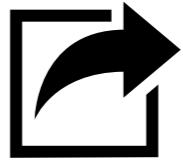
1

2

3

```
node.say('decision', observer, msg.data.offer);
```

Sending and Receiving Game Messages



[Game-Messages-v5](#)

When sending a message with any of the interactive methods (say, set, get, done), an object of class Game Message is created

1

2

3

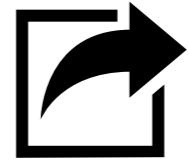
```
node.say('decision', observer, msg.data.offer);
```



```
GameMsg {  
    session: "2105497098240851",  
    created: "2019-11-24T20:46:41.318Z",  
    action: "say",  
    target: "DATA",  
    text: "decision", 1  
    to: "584498971106495", 2  
    data: 1, 3  
    from: "myexperiment",  
    sid: undefined,  
    stage: null,  
    id: 81124,  
    forward: 0,  
    priority: 0,  
    reliable: 1  
}
```

The ID of the logic is hidden with the name of the channel

Sending and Receiving Game Messages



[Game-Messages-v5](#)

When sending a message with any of the interactive methods (say, set, get, done), an object of class Game Message is created

1

2

3

```
node.say('decision', observer, msg.data.offer);
```



```
node.on.data("offer", function(msg){  
    // msg.data = 1;  
    var offer = "The offer is: " + msg.data;  
    // Display offer using W.  
    W.setInnerHTML("decision", offer);  
    node.done();  
});
```

```
GameMsg {  
    session: "2105497098240851",  
    created: "2019-11-24T20:46:41.318Z",  
    action: "say",  
    target: "DATA",  
    text: "decision", 1  
    to: "584498971106495", 2  
    data: 1, 3  
    from: "myexperiment",  
    sid: undefined,  
    stage: null,  
    id: 81124,  
    forward: 0,  
    priority: 0,  
    reliable: 1  
}
```

The ID of the logic is hidden with the name of the channel

Hands On Exercises



1. Tit-for-Tat Bot
2. Timeup Property
3. Observer's Reply

A Tit-for-Tat Bot



Let's build a bot that plays the **TIT-for-TAT strategy**.

If the bot goes first as dictator, it makes a *nice* offer of splitting the money 50-50.

If the bot goes first as observer, it repeats the last offer received as a dictator.

Original Bot Structure: Game Step

```
stager.extendStep('game', {
  roles: {
    DICTATOR: {
      cb: function() {
        this.node.done({ offer: 1 });
      }
    },
    OBSERVER: {
      cb: function() {
        // Store a local reference of node.
        var node = this.node;
        node.on.data('decision', function(msg) {
          setTimeout(function() {
            node.done();
          }, 5000);
        });
      }
    }
  })
});
```

node.on.data is waiting for a message from the server

When a message arrives which is labeled 'decision', the callback function is executed with the message as input parameter.

A more complex bot in a more complex game could make use of this information to send a reply accordingly. *Let's do it!*

A Tit-for-Tat Bot

Let's build a bot that plays the **TIT-for-TAT strategy**.

If the bot goes first as dictator, it makes a *nice* offer of splitting the money 50-50.

If the bot goes first as observer, it repeats the last offer received as a dictator.

OBSERVER

```
cb: function() {
  var node = this.node;
  node.on.data('decision', function(msg) {
    // Store previous offer.
    node.game.offer = msg.data;
    setTimeout(function() {
      node.done();
    }, 5000);
  });
}
```

As *observer*, the bot needs to store a reference to the last offer received.

node.game is the right place, because it is persistent across steps.

Note! The variable `msg` is a "**game message**" sent by `node.say`. We will later see how game messages look like inside.

A Tit-for-Tat Bot

Let's build a bot that plays the **TIT-for-TAT strategy**.

If the bot goes first as dictator, it makes a *nice offer* of splitting the money 50-50.

If the bot goes first as observer, it repeats the last offer received as a dictator.

DICTATOR

```
cb: function() {
    var node = this.node;
    // Did we receive an offer already?
    if ('undefined' === typeof node.game.offer) {
        node.done({ offer: 50 });
    }
    else {
        node.done({ offer: node.game.offer });
    }
}
```

As *dictator*, the bot needs to check if a previous offer was saved.

If not, it makes the first **nice offer** of 50.

Otherwise, repeat the last offer.

Understanding JavaScript Types

In the previous code example, we used `'undefined' === typeof node.game.offer` to know if a variable was previously defined. `typeof` returns the type of a variable in a string.

```
typeof 'John';          // 'string'  
typeof 1234;           // 'number'  
typeof function() {...} // 'function'  
typeof { foo: 'bar' }   // 'object'
```

Understanding JavaScript Types

In the previous code example, we used `'undefined' === typeof node.game.offer` to know if a variable was previously defined. `typeof` returns the type of a variable in a string.

```
typeof 'John';          // 'string'  
typeof 1234;           // 'number'  
typeof function() {...} // 'function'  
typeof { foo: 'bar' }   // 'object'
```

However, **arrays** are also considered objects

```
typeof [ 0, 1, 2, 3 ] // 'object'
```

Understanding JavaScript Types

In the previous code example, we used `'undefined' === typeof node.game.offer` to know if a variable was previously defined. `typeof` returns the type of a variable in a string.

```
typeof 'John';          // 'string'  
typeof 1234;           // 'number'  
typeof function() {...} // 'function'  
typeof { foo: 'bar' }   // 'object'
```

However, **arrays** are also considered objects

```
typeof [ 0, 1, 2, 3 ] // 'object'
```

If a variable has not yet been defined or assigned a value, its default type is **undefined**.

```
var aNewVariable;        // no value is assigned  
typeof aNewVariable      // 'undefined'  
aNewVariable = {};       // we assign an empty object  
typeof aNewVariable;     // 'object'  
typeof aNewVariable.foo; // 'undefined'
```

Understanding JavaScript Types

In the previous code example, we used '`undefined`' === `typeof node.game.offer` to know if a variable was previously defined. `typeof` returns the type of a variable in a string.

```
typeof 'John';          // 'string'  
typeof 1234;           // 'number'  
typeof function() {...} // 'function'  
typeof { foo: 'bar' }   // 'object'
```

It should be clear why:

```
typeof (typeof 1234); // 'string'
```

However, **arrays** are also considered objects

```
typeof [ 0, 1, 2, 3 ] // 'object'
```

To know if a variable is an array you can use:

```
J.isArray();
```

Remember! The `J` object is created by `nodeGame` and it contains several helper methods.

If a variable has not yet been defined or assigned a value, its default type is **undefined**.

```
var aNewVariable;          // no value is assigned  
typeof aNewVariable        // 'undefined'  
aNewVariable = {} ;        // we assign an empty object  
typeof aNewVariable;       // 'object'  
typeof aNewVariable.foo;   // 'undefined'
```

The **timeup** step-property

The **timeup** step-property determines what happens when the timer runs out

The default behavior is to call **node.done** and go to the next step

In some steps, the *user need to take a decision before advancing*, so you must code what happens in this case

In **group-behavior experiments**, it is recommended that you keep the game flowing and avoid any further delay, which might cost other people to dropout

A standard policy is: *repeat the last decision, or make a random one if none is available*. Let's implement it!

The **timeup** step-property for the DICTATOR

```
cb: function() {
    // Note! Some code has been omitted for space limitation.
    button.onclick = function() {
        // When the user clicks the button, we store the decision under node.game
        node.game.decision = lastDecision;
        node.done({ offer: decision });
    };
},
timeup: function() {
    var n;
    if ('undefined' === typeof node.game.decision) {
        n = J.randomUUID(-1, 100);
    }
    else {
        n = node.game.decision;
    }
    W.getElementById('offer').value = n;
    W.getElementById('submitOffer').click();
}
```

The timeout step-property for the DICTATOR

```
cb: function() {
    // Note! Some code has been omitted for space limitation.
    button.onclick = function() {
        // When the user clicks the button, we store the decision under node.game
        node.game.decision = lastDecision;
        node.done({ offer: decision });
    };
},
timeout: function() {
    var n;
    if ('undefined' === typeof node.game.decision) {
        n = J.randomUUID(-1, 100);
    }
    else {
        n = node.game.decision;
    }
    W.getElementById('offer').value = n;
    W.getElementById('submitOffer').click();
}
```

Random decision.

Previous decision.

We update the display, by inserting the value inside the input form.

We click the button to trigger the onclick listener, which will store the offer and call node.done.

Let the Observer Reply



- Modify 'game' step:
 - Observer replies with a text message to approve or disapprove offer

Prepare HTML Elements



We begin by modifying the HTML file **public/game.htm**

```
<div id="observer" style="display: none">
  <h4 id="waitFor">
    Waiting for the decision of the dictator<span id="dots"></span>
  </h4>
  <span id="decision"></span>
  <p style="display: none" id="reply">
    Tell the dictator what you think
    <input type="text" id="observer_reply" />
    <button id="submit_reply">Submit</button>
  </p>
  <br/>
</div>
```

Prepare HTML Elements



We begin by modifying the HTML file **public/game.htm**

```
<div id="observer" style="display: none">
  <h4 id="waitFor">
    Waiting for the decision of the dictator<span id="dots"></span>
  </h4>
  <span id="decision"></span>
  <p style="display: none" id="reply">
    Tell the dictator what you think
    <input type="text" id="observer_reply" />
    <button id="submit_reply">Submit</button>
  </p>
  <br/>
</div>
```

These elements are initially **hidden**. We will show them with JavaScript at the right time.

Always add ids to make it easy to fetch these elements in JavaScript



Add Observer Code



Observer



```
node.on.data("decision", function(msg) {
    // Display offer using W.
    W.setInnerHTML("decision", "The dictator offered: " + msg.data);
    // Show hidden part.
    W.show("reply");
    // Add listener that sends the text back to dictator.
    W.gid("submit_reply").onclick = function() {
        // Say message directly to dictator.
        node.say("reply", "dictator_id", W.gid("observer_reply").value);
        // End step.
        node.done();
    };
}) ;
```



Add Observer Code



Observer



```
node.on.data("decision", function(msg) {  
    // Display offer using W.  
    W.setInnerHTML("decision", "The dictator offered: " + msg.data);  
    // Show hidden part.  
    W.show("reply"); Shows the button and the input form to reply  
    // Add listener that sends the text back to dictator.  
    W.gid("submit_reply").onclick = function() {  
        // Say message directly to dictator.  
        node.say("reply", "dictator_id", W.gid("observer_reply").value);  
        // End step.  
        node.done(); Sends the reply to the dictator with node.say.  
    };  
});
```

Recipient ID for node.say

```
node.say("reply", "dictator_id", w.gid("observer_reply").value);
```



How do I know the ID of the recipient of a message?

For security reason, only the logic has all the IDs.

So, the ID needs to be sent from the logic to the client if needed. There are a few options how to do it.

In this example, we are using the Matcher API to match participants in pairs and assign them roles (dictator and observer). When the matcher API is used, the ID can be found under **node.game.partner**

```
node.say("reply", node.game.partner, w.gid("observer_reply").value);
```

Recipient ID for node.say

```
node.say("reply", "dictator_id", w.gid("observer_reply").value);
```



How do I know the ID of the recipient of a message?

For security reason, only the logic has all the IDs.

So, the ID needs to be sent from the logic to the client if needed. There are a few options how to do it.

In this example, we are using the Matcher API to match participants in pairs and assign them roles (dictator and observer). When the matcher API is used, the ID can be found under **node.game.partner**

```
node.say("reply", node.game.partner, w.gid("observer_reply").value);
```

Or, even simpler, here we are replying to an incoming msg, so we can use the **msg.from** property.

```
node.say("reply", msg.from, w.gid("observer_reply").value);
```



Add Dictator Code



Dictator



```
submit.onclick = function() {  
  
    // We send offer directly to the observer,  
    // and we delay calling node.done.  
    node.say("decision", node.game.partner, offer);  
};  
  
// We wait for the reply, we display it, and  
// we call node.done (with a random timer).  
node.on.data("reply", function(msg) {  
    W.writeln("Your partner thinks of your offer: " + (msg.data || "nothing"));  
    node.timer.randomDone();  
});
```

Add Dictator Code



Dictator



```
submit.onclick = function() {  
  
    // We send offer directly to the observer,  
    // and we delay calling node.done.  
    node.say("decision", node.game.partner, offer);  
};  
  
// We wait for the reply, we display it, and  
// we call node.done (with a random timer).  
node.on.data("reply", function(msg) {  
    W.writeln("Your partner thinks of your offer: " + (msg.data || "nothing"));  
    node.timer.randomDone();  
});
```

Same labels as in Observer: "decision" and "reply"

Logical OR, shortcut to assign a default value, if the first term is "falsy." Hint!
Undefined if falsy, remember the comparison tables with two and three equals?

Client Type logic.js



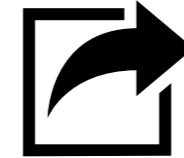
- Contains code that will be executed on the server
 - Has access to the nodeGame client API AND to all server objects

Client Type logic.js



- Contains code that will be executed on the server
 - Has access to the nodeGame client API AND to all server objects
- What does it do?
 - It waits for all players to finish a step, if there is nothing better to do
 - Make sure the timers are respected
 - Can push/kick out non-responsive players
 - It receives incoming messages from players and stores them (`node.game.memory`)
 - It matches partners, and assigns roles

Client Type logic.js



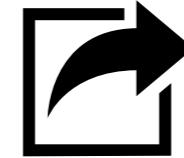
[Matching-Roles-Partners-v5](#)



Logic Client Type

```
stager.extendStep('bidder', {
  matcher: {
    // Available roles.
    roles: [ 'BIDDER', 'RESPONDENT' ],
    // Each player is matched with a
    // partner and is assigned a role.
    match: 'roundrobin',
    // How to continue matching after
    // all combinations are exhausted
    cycle: 'repeat_invert'
  }
});
```

Matcher API



[Matching-Roles-Partners-v5](#)



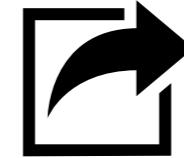
Logic Client Type

```
stager.extendStep('bidder', {
  matcher: {
    // Available roles.
    roles: [ 'BIDDER', 'RESPONDENT' ],
    // Each player is matched with a
    // partner and is assigned a role.
    match: 'roundrobin',
    // How to continue matching after
    // all combinations are exhausted
    cycle: 'repeat_invert'
  }
});
```

Player Client Type

```
stager.extendStep('bidder', {
  roles: {
    BIDDER: {
      frame: 'bidder.htm',
      cb: function() {
        // Bidder stuff.
      }
    },
    RESPONDENT: {
      frame: 'resp.htm',
      cb: function() {
        // Respondent stuff.
      }
    }
  }
});
```

Client Type logic.js



[Matching-Roles-Partners-v5](#)



Logic Client Type

```
stager.extendStep('bidder',  
  matcher: {  
    // Available roles.  
    roles: [ 'BIDDER', 'RESPONDENT' ],  
    // Each player is matched with a  
    // partner and is assigned a role.  
    match: 'roundrobin',  
    // How to continue matching after  
    // all combinations are exhausted  
    cycle: 'repeat_invert'  
  }  
);
```

Logic lists roles
Player implements

Player Client Type

```
stager.extendStep('bidder', {  
  roles: {  
    BIDDER: {  
      frame: 'bidder.htm',  
      cb: function() {  
        // Bidder stuff.  
      }  
    },  
    RESPONDENT: {  
      frame: 'resp.htm',  
      cb: function() {  
        // Respondent stuff.  
      }  
    }  
  }  
);
```

Custom Group Matching: Logic



```
// Let us assume a 4 player game, and we want 2 groups of two.  
  
// Returns an array of ids.  
var ids = node.game.playerList.id.getAllKeys();
```

Custom Group Matching: Logic



```
// Let us assume a 4 player game, and we want 2 groups of two.  
  
// Returns an array of ids.  
var ids = node.game.playerList.id.getAllKeys();  
  
// Make two random groups.  
var groups = J.getNGroups(ids, 2);
```

Custom Group Matching: Logic



```
// Let us assume a 4 player game, and we want 2 groups of two.  
  
// Returns an array of ids.  
var ids = node.game.playerList.id.getAllKeys();  
  
// Get a copy of original array.  
var ids2 = ids.slice();  
node.say('yourpartner', ids[0], ids2.splice(0, 1));  
  
// Make two random groups.  
var groups = J.getNGroups(ids, 2);  
  
// Loop through the groups and assign partners.  
for (var i = 0; i < groups.length; i++) {  
    // Get array of group member ids.  
    var members = groups[i];  
    // Tell each member up me  
    node.say('yourpartner', members[0], members[1]);  
    // Recipient ID is Payload (Partner ID)
```

Custom Group Matching: Player



```
node.on.data('yourpartner', function(msg) {  
    // Store partner ID for later use.  
    node.game.mypartner = msg.data;  
    // Continue with the rest of the step.  
});
```

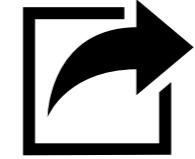
If creating all the groups takes long (e.g., if the group size is very large), it makes sense to have custom grouping in an own step and call **node.done** as soon as the group information arrives.

Custom Group Matching: Player



```
// Optional step for custom grouping.  
// Note! This step is skipped in game.stages, and these groups are not  
// actually used in the game step. Do you know how to adjust it?  
stager.extendStep('grouping', {  
    widget: {  
        name: 'ContentBox',  
        options: {  
            mainText: 'Please wait while groups are being formed'  
        }  
    },  
    cb: function() {  
        node.on.data('yourpartner', function(msg) {  
            node.game.mypartner = msg.data;  
            console.log("My partner is: " + msg.data);  
            node.timer.randomDone();  
        }) ;  
    }  
}) ;
```

Memory Database



[Memory-Database-v5](#)



node.game.memory

```
stager.extendStep('end', {
  cb: function() {
    node.game.memory.save('data.json');
  }
});
```

Get Your Data



```
stager.extendStep('end', {
  cb: function() {
    node.game.memory.save('data.json');
  }
});
```

The screenshot shows the logic.js application interface with two main sections: 'Data Folder' and 'Memory Database'.

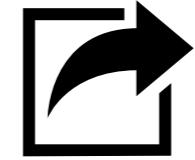
Data Folder: This section displays the contents of a data folder. It includes a 'Refresh' button, a dropdown menu, and tabs for 'Clients', 'Settings', 'Waitroom', 'Auth', 'Requirements', 'Results', and 'Server'. The 'Results' tab is currently selected. Below these are two panels: 'Data Folder' and 'Memory Database'.

- Data Folder Panel:** Shows the last modified date (Mon Nov 25 2019 03:02:32 GMT+0100) and provides a 'Refresh' button and a link to 'Download all in a zip archive'.
- Memory Database Panel:** Provides a warning about the performance impact of downloading a high-volume database and contains a link to 'Download the memory database of all game rooms'.

Data Folder Content Table:

File	Modified
room000005\data.json	2019-11-25T01:44:18.551Z
room000010\data.json	2019-11-25T01:53:26.931Z

Memory Database



[Memory-Database-v5](#)



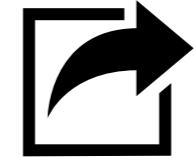
`node.game.memory`



`node.done({ offer: 1 });`

```
{  
  stage: { stage: 2,  
           step: 1,  
           round: 1 },  
  player: "id12345678",  
  timestamp: 1480336721939  
  offer: 1,  
  timeup: false,  
  time: 4321, // Time passed from the beginning of the step  
  done: true  
};
```

Memory Database



[Memory-Database-v5](#)



logic.js

node.game.memory



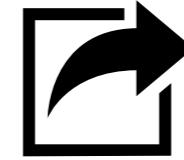
```
{  
  stage: { stage: 2,  
           step: 2,  
           round: 1 },  
  player: "id12345678",  
  timestamp: 148033845673  
  foo: "bar" ,  
};
```



player.js

```
// Set an object in memory explicitly.  
node.set({ foo: "bar" });
```

Memory Database



[Memory-Database-v5](#)

```
var m = node.game.memory;
```

```
// Lazy evaluation: add a bunch of statements
// and evaluates all of them at the end.
m.select("player", "=", "id12345678");
m.and("stage.step", "=", 1)
  .and("offer") // Must have property "offer"
  .fetch() [0]; // Returns array, get first element
```

```
// Use indexes.
```

```
// By sender.
```

```
m.player["id12345678"].last();
```

```
// By stage.
```

```
m.stage["2.1.1"].select("offer").fetch() [0];
```



Self Exercise



- Explore this RunKit on selecting and fetching items from `node.game.memory`
 - <https://runkit.com/shakty/gamedb>