CA110
Space API

David Gray
Version 0.2

26th February 2015

Contents

1 HTTP Requests

For the HTTP/JSON APIs, all requests use HTTP GET.

2 Common JSON Objects

2.1 3-D Coordinates

A **3-D Coordinates** object is a JSON object with the following fields:

`x`: A real number.

`y`: A real number.

`z`: A real number.

For example:

```
{
    "x":42363.5374374,
    "y":3947394796.215,
    "z":846.26732
}
```

For **positions**, the coordinates measure **light-years** and **directions** measure **radians**.

2.2 Player's Details

A **Player's Details** object is a JSON object with the following fields:

`name`: The name of the player.

`ship`: The name of the player's ship.

`position`: The players position as a 3-D Coordinates object.

`direction`: The players direction as a 3-D Coordinates object.

For example:

```
{
    "name":"Master Yoda",
    "ship":"astratis_v1",
    "position": {
        "x": 626246,
        "y": 23526.2664,
        "z": 25.125
    },
    "direction": {
        "x":0.2,
        "y":1.4,
        "z":0
    }
}
```

2.3 Inventory

A **Inventory** object is a JSON object with the following fields:

`id`: An integer.

`amount`: An integer.

For example:

```
    {
        "id":5,
        "amount":234
    }
```

3 Responses

All HTTP/JSON requests respond with a JSON object containing a boolean `success` field that has one of two values:

1. `true`: The request has been successful and the JSON object will have other fields containing the results of the request.
2. `false`: The request has failed and the JSON object will have two other fields:

   `code`: A integer value as described in Table 1.

   `errorText`: A string giving a text message for the error.

For example,

```
{
    "success":false,
    "code":101,
    "errorText":"Missing id parameter in http request"
}
```

Code	**Reason**
100	Other error
101	Missing parameter in request
102	Unknown parameter in request
103	Unknown request
104	Server not ready or busy
105	Not players turn
106	Authentication failure
107	Sender not the originator of chat message
108	Unknown player
	. . .

Table 1: Error Codes

4 Discovery API

4.1 `getServers` request

4.1.1 Parameters

None

4.1.2 Response fields

`addresses`: A JSON object containing the URLs of the three servers:

`authServer` : URL string
`gameServer` : URL string
`tradeServer` : URL string

4.1.3 Semantics

The addresses of the servers to be used.

4.1.4 Example Exchange

Request: `http://???/getServers`

Response:

```
{
    "success":true,
    "addresses": {
        "authServer":  "https://1.1.1.1:3000",
        "gameServer":  "https://2.2.2.2:3001",
        "tradeServer": "https://2.2.2.2:3002"
    }
}
```

5 Authentication API

5.1 `register` request

Allows a used to register with the API.

TBD.

5.2 `authenticate` request

Allows an exiting user to obtain an authentication token.

TBD.

5.3 `version` request

5.3.1 Parameters

None

5.3.2 Response fields

`major`: The major version of the API.

`minor`: The minor version of the API.

5.3.3 Semantics

API version information

5.3.4 Example Exchange

Request: `http://???/version`

Response:

```
{
    "success":true,
    "major":0,
    "minor":2
}
```

6 Trade API

TBD.

7 Game API

The Game API is implemented on a bidirectional, stream-oriented connection[1] over which messages are transferred. Each message has a **name** and a **content**.

1. A message's name is a string that identifies the **category** of message.
2. A message's content is a JSON object[2].

7.1 Connection Handshake

We could use a single response message with a boolean flag, but using two different message categories should facilitate neater code. Please comment.

When a connection is established[3] the client must send a **start** message and *must not send any further message until it receives an **accepted** message from the server.* If the sever returns a **rejected** messages then the client should terminate.

7.1.1 start message

The JSON object for a **start** message has the following fields:

`name`: The user's name.

`token`: The user's authentication token.

For example:

```
{
    "name":"Hans Solo",
    "token":"98786vs8g5bsg875w6g57gdg"
}
```

[1]The connection must support the transfer of JSON objects belonging to different categories. For example, `engine.io.protocol` over `TCP` is a suitable protocol.

[2]Primitive JSON types and arrays cannot be used as a message's content.

[3]`engine.io.protocol` has its own handshake protocol used when a TCP connection is established.

7.1.2 accepted message

The JSON object for an **accepted** message has the following fields:

`major`: The major version of the API.

`minor`: The minor version of the API.

`position`: A 3-D Coordinates object.

`direction`: A 3-D Coordinates object.

For example:

```
{
    "major":0,
    "minor":2
    "position": {
        "x":42363.5374374,
        "y":3947394796.215,
        "z":846.26732
    },
    "direction": {
        "x":1.457,
        "y":0.525,
        "z":0.2546
    }
}
```

7.1.3 rejected message

The JSON object for an **rejected** message has the following fields:

`code`: A integer value as described in Table 1.

`errorText`: A string giving a text message for the error.

For example:

```
{
    "code":106,
    "errorText":"Expired Token"
}
```

7.2 Chatting

I have made some changes to chat. In particular, I'm proposing that chat messages are not acknowledged, unless there is an error and that the name of the originator of messages is included by the originator. Please comment.

There are two categories of chat message:

1. **gchat** - global chat messages.
2. **pchat** - private chat messages.

Both categories of chat message are sent from an originator's client to the server and from the server to the recipient(s). The server does not acknowledge chat messages unless there is an error, in which case, the server sends a **failedChat** message to the originator.

Once a connection has been accepted, the client may send chat messages to the server and should be prepared to accept chat message from the server.

7.2.1 gchat message

The JSON object for a **gchat** message has the following fields:

`time`: Unix timestamp in milliseconds.

`originator`: The originator's name.

`errorText`: The chat text.

For example:

```
{
    "time":368389679893479,
    "originator":"Master Yoda",
    "text":"Welcome to Dagobah"
}
```

7.2.2 pchat message

We could change the `recipient` field to `recipients` with a list of names. Note that this would make the **chatFailure** message a bit more complex.

The JSON object for a **pchat** message has the following fields:

`time`: Unix timestamp in milliseconds.

`originator`: The originator's name.

`recipient`: The recipient's name.

`text`: The chat text.

For example:

```
{
    "time":368389679893492,
    "originator":"Master Yoda",
    "recipient":"Hans Solo",
    "text":"Welcome to Dagobah"
}
```

7.2.3 chatFailure message

The JSON object for a **chatFailure** message has the following fields:

`code`: A integer value as described in Table 1.

`message`: A string giving a text message for the error.

`original`: A copy of the original chat message.

For example:

```
{
    "code":107,
    "text":"Sender not originator",
    "original": {
            "time":368389679893492,
            "originator":"Master Yoda",
            "recipient":"Hans Solo",
            "errorText":"Welcome to Dagobah"
    }
}
```

7.3 Time

How frequently will these messages be sent and exactly what will the time values be used for? Could we not simply return the time once in the **accepted** message?
Please comment.

Once a connection has been accepted, the server will periodically send a **time** message to each client.

The JSON object for a **time** message has the following fields:

`time`: Unix timestamp in milliseconds.

For example:

```
{
    "time":368389679893479
}
```

7.4 Other Players

I'm not sure what is required here. Given that we want a game with massively many players, it seems unreasonable to periodically send information for all players. So there are a number of questions:

1. **When** do we send this information? For example, do we only send it when a player moves?
2. For a given client, **which** players do we send? For example, do we only send players **near** to the client?
3. Do we want the client to ask explicitly for this information? Or alternatively, do we want the client to "register" players of interest?

Please comment.

Once a connection has been accepted, the server will ~~periodically~~ send **otherPlayers** messages to each client.

The JSON object for an **otherPlayers** message has the following fields:

`players`: An array of Player's Details objects.

For example:

```
{
    "players":[
        {
            "name":"Master Yoda",
            "ship":"astratis_v1",
            "position": {
                "x": 626246,
                "y": 23526.2664,
                "z": 25.125
            },
            "direction": {
                "x":0.2,
                "y":1.4,
                "z":0
            }

        },
        . . .
    ]
 }
```

7.5 Move

Do we really need to send this information periodically? Could we just send it when a player moves?
Please comment.

Once a connection has been accepted, each client will ~~periodically~~ send **move** messages to the server.

The JSON object for an **move** message has the following fields:

`position`: A 3-D Coordinates object.

`direction`: A 3-D Coordinates object.

For example:

```
{
    "position": {
        "x": 626246,
        "y": 23526.2664,
        "z": 25.125
    },
    "direction": {
        "x":0.2,
        "y":1.4,
        "z":0
    }
 }
```

7.6 Inventory

More work is needed on how the items are represented and what data is stored. Is an integer the correct type of id? For sprint #2 I do not see any use of this message type. In particular, there is insufficient information to display inventory details to a user, e.g., items have no name and there is no way of getting names.

What triggers these messages being sent? Presumably a UI event could trigger a client to send an **inventory** message, but when does a server send an **inventory** message?

Should **inventory** messages be part of the trade API?

Please comment.

Once a connection has been accepted, the client and server may send **inventory** messages.

The JSON object for an **inventory** message has the following fields:

`items`: An array of inventory objects.

For example:

```
{
    "items": [
        {
            "id":5,
            "amount":1
        },
        {
            "id":1,
            "amount":1005627624
        }
    ]
 }
```