# frontend API (contained in Middleware folder) docs

## HTTP_API.ts

This class is used throughout the project to send HTTP requests (ex: the initial request to join a game).

This class is static and cannot be manually instantiated. This ensures consistency with the `WS_API` (discussed later) and avoids potential confusion with multiple sources of transmitting HTTP requests and responses.

This class extends from the `Base_HTTP_API.ts` class which is located in the `src/shared/` folder. This base class defines all the underlying framework for HTTP communication for both the client and game nodes. The `HTTP_API.ts` file that is then stored in the individual nodes implements the actual requests that are relevant to each node.

For a detailed explanation of each function, please see the inline JSDoc-formatted comments that are placed before each method.

## WS_API.ts

This class is used throughout the project to manage the WebSocket connection that is maintained with the server as well as transmit and receive game state data. WebSockets are the primary method of bidirectional communication in this project, so each client node maintains a persistent connection with the server.

This class is static and cannot be manually instantiated. This ensures that there is never more than one WebSocket connection with the server at any time.

This class extends from the `Base_WS_API.ts` class which is located in the `src/shared/` folder. This base class defines all the underlying framework for setting up, maintaining, using, and destroying the WebSocket connection for both the client and game nodes. The `WS_API.ts` file that is then stored in the individual nodes implements the actual requests that are relevant to each node.

#### Some important notes about using this class

- Although the WebSocket API does not support async/await calling paradigm, we have retrofitted it and so all the methods in the class can be used with `await` to await completion before continuing.
- Before you can send or receive data, you must call the `setupWebSocketConnection` method to establish a connection to the server. After the returned promise completes, the connection will be ready for data transmission.
- You can register a listener that will be called when a message is received from the server via the `addIncomingMessageCallback` method.
  - The system allows for multiple callbacks and thus allows us to split the handling of messages to their respective views. Thereby preserving Separation of Concerns.
  - It is important to unregister the callbacks via the `removeIncomingMessageCallback` method before closing the view though to avoid sending data to non-existant places.
- Each request originating from the client or game nodes will have a `requestId` this is used to associate requests with their responses. Therefore, we can handle out of order messages as well as many duplicate requests without issue.

For a detailed explanation of each function, please see the inline JSDoc-formatted comments that are placed before each method.

# File Structure of frontend nodes

## Views

The views folder contains individual "screens" that are displayed throughout the webapp. For example, the game join screen or the answer a multiple-choice question screen.

Each view is composed of:

- A script - used for handling the logic (in TypeScript)
- A template - HTML that does the layout of the view
- A style - CSS that does micro-adjustments to the layout to make a more visually appealing view.

Each view is self-contained inside a single `.vue` file although it can reference components, middleware, and assets.

### Components

The components folder contains views that are used in several places as part of a larger view. For example, a timer UI adornment.

Components follow the exact same pattern as views and have all the same limitations and flexibilities as regular views. Components also have the `.vue` file extension.

### Assets

The assets folder contains images and other resource files that are used throughout the webapp.

### Middleware

The middleware folder contains all the API-related typescript logic that is references throughout the webapp. The files in this section act as an abstraction layer to abstract the requests and make them all as easy to use as possible. See the frontend API section of this file for specifics about how the files in this folder function.

### App.vue

Both the game and client nodes have an `App.vue` file. This is the main view file that selects which sub-view to show. The `App.vue` file runs throughout the entire execution of the webapp and handles the following items:

- View selection and navigation
- Some incoming WebSocket message handling (this is split between the sub-views as well as `App.vue` handles the navigation related ones).
- game state storage and manipulation (since `App.vue` is always executing, it is a good place to store all the game state)

### Main.ts

Both the client and game nodes have this file. It is used as an entry point to load the `App.vue` file at runtime. It can also be used to register listeners and other long-running processes that should not be linked to any particular view.

### ../src/shared

This folder contains any files that are shared between nodes (client, server. or game). Most of the subfolders contains typing information (since TypeScript separates types from the data) but there is also the `Base_WS_APi.ts` and `Base_HTTP_API.ts` files which handle the communication between frontend nodes and the server (See the frontend API section for more information).

## Client Node

Each run of the game will consist of 2 or more client nodes. Each client node corresponds to one player.

### Views

#### HomeView.vue

This view is responsible for collecting user input (in the form of a game code and username) and then transmitting it to the server to join a game. Once a game is joined, this view changes to the `MainView`.

#### MainView.vue

This view is responsible for showing the current leaderboard standings to the user as well as who is the next player. This view is displayed while it is not the player's turn to answer a question or acknowledge a consequence after they have joined a game.

#### MultipleChoiceQuestion.vue

This view is shown to the player once it is their turn to answer a question. Upon answering a question, the player's response is captured and sent to the server. Then the client node returns to the `MainView`.

# Game node

## Components

### Board.vue

Displays the positional of each player as a coloured circle on a virtual board. Once the players reach the last position, they win. Each player's circle color is randomly assigned but is consistent through all the nodes.

### ConsequenceModal.vue

This component pops-up when a player must face a consequence. It displays the consequence in textual form and shows a timer. After the timer expires, the component disappears and the game continues.

## PlayersList.vue

This view lists all the players that are currently in the game as well as highlights the player who is currently answering a question.

## Views

### HomeView.vue

This view is responsible for creating and then starting a game, displaying the game code (once a game is created) so that the players can enter the code on the client nodes, listing the already joined players, and configuring the game. After all that is completed, the game node switches to the `MainView`.

### MainView.vue

This is the main view that is displayed throughout the game. It consists of the board (which shows the current location of each player) as well as a player list (which shows the current players and highlights the player whose turn is next).

### MultipleChoiceQuestion.vue

This view appears when a client node needs to answer a question. This view shows the question as well as a related or decorational background image as well as a timer while the player is answering. After the player answer, or the timer runs out, the gam enode transitions back to the `MainView`.

### FinalStandings.vue

This view appears after a game has ended. It shows the top placing players.

# Server Node

## Module: controllers/AuthController

### Functions

### generateJWT

- **generateJWT**(`requestData`): `Promise<string>`

Generate the user access token that identifies each connected user, and create their user within the database.

### Parameters

| Name | Type | Description |
|---|---|---|
| `requestData` | `Object` | The username, game code, and type of user in which is generating their authenticat |
| `requestData.color?` | `Color` | If the user is a client node user, they will provide the color for their player piece. |
| `requestData.gameCode` | `string` | The game code for the game that the user is going to be related to. |
| `requestData.userType` | `"Game" \| "Client"` | Whether the user being created is a game or client node user. |
| `requestData.username` | `string` | The username of the user sending te request to generate a JWT token. |

**Returns** `Promise<string>`

Resolution code with JWT embedded.

**Defined in** controllers/AuthController.ts:23

## Module: controllers/ClientController

### Functions

#### joinGame

- **joinGame**(`req`, `res`): `Promise<unknown>`

Allow user to join a game assuming they provide their username and the game code.

**Parameters**

| Name | Type | Description |
|---|---|---|
| `req` | `FastifyRequest<{ Body: { game_code: string ; username: string } }, RawServerDefault, IncomingMessage, FastifySchema, FastifyTypeProviderDefault, unknown, FastifyBaseLogger, ResolveFastifyRequestType<FastifyTypeProviderDefault, FastifySchema, { Body: { game_code: string ; username: string } }>>` | The user request containing their username and the game id. |
| `res` | `FastifyReply<RawServerDefault, IncomingMessage, ServerResponse<IncomingMessage>, RouteGenericInterface, unknown, FastifySchema, FastifyTypeProviderDefault, unknown>` | The response to indicate to the user whether that their request succeeded. |

**Returns** `Promise<unknown>`

A resolution, or rejection, to indicate if the request was successful.

**Defined in** controllers/ClientController.ts:21

## Module: controllers/GameController

### Functions

#### checkWinner

- **checkWinner**(`gameID`): `Promise<string | boolean>`

Check if any players are in the winner state.

**Parameters**

| Name | Type | Description |
|---|---|---|
| `gameID` | `string` | The game code string for the game you want to check the winner of. |

**Returns** `Promise<string | boolean>`

**Defined in** controllers/GameController.ts:693

---

#### createGame

- **createGame**(`req`, `res`): `Promise<unknown>`

Creates a game object from an incoming request.

**Parameters**

| Name | Type | Description |
|---|---|---|
| req | FastifyRequest<{ Body: { theme_pack: string } }, RawServerDefault, IncomingMessage, FastifySchema, FastifyTypeProviderDefault, unknown, FastifyBaseLogger, ResolveFastifyRequestType<FastifyTypeProviderDefault, FastifySchema, { Body: { theme_pack: string } }»</td> | Incoming request object from the game node. |
| res | FastifyReply<RawServerDefault, IncomingMessage, ServerResponse<IncomingMessage>, RouteGenericInterface, unknown, FastifySchema, FastifyTypeProviderDefault, unknown> | Outgoing response handler. |

**Returns** Promise<unknown>

Returns a response wrapped in a promise to be handled by the Fastify router.

**Defined in** controllers/GameController.ts:99

---

**handleConsequence**

- **handleConsequence**(connections, game, data, early): Promise<void>

Handle consequence timeout or ending early.

**Parameters**

| Name | Type | Description |
|---|---|---|
| connections | Object | The websocket information of all players connected to the specific game. |
| connections.clients | ClientConn[] | - |
| connections.host | ClientConn | - |
| connections.turn? | Object | - |
| connections.turn.movement_die | number | - |
| connections.turn.timeout? | Timeout | - |
| connections.turn.turn_start | number | - |
| game | PopulatedGame | The populated game instance to fetch information about the current gam |
| data | WebsocketRequest | Information related to the request, such as request id. |
| early | boolean | Is this request ending the game before the timeout? |

**Returns** Promise<void>

This is a mutation function in which modifies the next game state and sends it to the players.

**Defined in** controllers/GameController.ts:630

---

**nextPlayer**

- **nextPlayer**(gameID): Promise<string>

Given a game id, prepare to start the game. To do so: 1. Randomize the player array to determine turn order. 2. Change the boolean in the game model to be True. 3. Return the username of the first player in the turn order.

**Parameters**

| Name | Type | Description |
|---|---|---|
| gameID | string | The Model Game ID within the database. |

**Returns** `Promise<string>`

The username of the player next in the rotation.

**Defined in** controllers/GameController.ts:231

---

**questionAnswer**

- **questionAnswer**(`connections`, `data`, `username`, `game`): `Promise<boolean>`

Handle a user sending an answer request to the server

**Parameters**

| Name | Type | Description |
|---|---|---|
| `connections` | `Object` | The websocket information of all players connected to the specific game. |
| `connections.clients` | `ClientConn[]` | - |
| `connections.host` | `ClientConn` | - |
| `connections.turn?` | `Object` | - |
| `connections.turn.movement_die` | `number` | - |
| `connections.turn.timeout?` | `Timeout` | - |
| `connections.turn.turn_start` | `number` | - |
| `data` | `WebsocketRequest` | Information related to the request, such as request id and the question a |
| `username` | `string` | The username of the user who send the websocket request. |
| `game` | `PopulatedGame` | The populated game instance to fetch information about the current gam |

**Returns** `Promise<boolean>`

Whether the answer submitted is, or is not, correct.

**Defined in** controllers/GameController.ts:496

---

**questionEnd**

- **questionEnd**(`connections`, `game`, `data`, `early`): `Promise<void>`

The question has ended, either by timeout or by answer. Handle accordingly.

**Parameters**

| Name | Type | Description |
|---|---|---|
| `connections` | `Object` | The websocket information of all players connected to the specific game. |
| `connections.clients` | `ClientConn[]` | - |
| `connections.host` | `ClientConn` | - |
| `connections.turn?` | `Object` | - |
| `connections.turn.movement_die` | `number` | - |
| `connections.turn.timeout?` | `Timeout` | - |
| `connections.turn.turn_start` | `number` | - |
| `game` | `PopulatedGame` | The populated game instance to fetch information about the current gam |
| `data` | `WebsocketRequest` | Information related to the request, such as request id. |
| `early` | `boolean` | Is this request ending the game before the timeout? |

**Returns** `Promise<void>`

This is a mutation function in which modifies the next game state and sends it to the players.

**Defined in** controllers/GameController.ts:561

---

**startGame**

- **startGame**(`gameID`): `Promise<string>`

Given a game id, prepare to start the game. To do so: 1. Randomize the player array to determine turn order. 2. Change the boolean in the game model to be True. 3. Return the username of the first player in the turn order.

**Parameters**

| Name | Type | Description |
|---|---|---|
| `gameID` | `string` | The Model Game ID within the database. |

**Returns** `Promise<string>`

The username of the player first in the rotation.

**Defined in** controllers/GameController.ts:193

---

**turn**

- **turn**(`connections`, `data`, `game`): `Promise<boolean>`

Handle the turn logic for a single round of the game, triggered by the game node sending a message.

**Parameters**

| Name | Type | Description |
|---|---|---|
| `connections` | `Object` | List of all Websockets relevant to the game that this turn is for. |
| `connections.clients` | `ClientConn[]` | - |
| `connections.host` | `ClientConn` | - |
| `connections.turn?` | `Object` | - |
| `connections.turn.movement_die` | `number` | - |
| `connections.turn.timeout?` | `Timeout` | - |
| `connections.turn.turn_start` | `number` | - |
| `data` | `WebsocketRequest` | Any relevant data that the game node sends across the websocket stream |
| `game` | `PopulatedGame` | The game state. We know that the sender of these messages is the game |

**Returns** `Promise<boolean>`

**Defined in** controllers/GameController.ts:271

## Module: controllers/QuizController

**Functions**

**formatConsequence**

- **formatConsequence**(`theme_pack_name`, `used_consequences`): `Promise<Consequence>`

Generate a consequence for the game.

**Parameters**

| Name | Type | Description |
|---|---|---|
| `theme_pack_name` | `string` | The name of the theme pack file. |
| `used_consequences` | `number[]` | List of already used consequence ids. |

**Returns** `Promise<Consequence>`

The consequence fetched for the game.

**Defined in** controllers/QuizController.ts:169

---

### formatQuestion

- **formatQuestion**(`theme_pack_name`, `category`, `question_type`, `used_questions`): `Promise<{ id: number ; media_type: null | "image" | "video" ; media_url: null | string ; options: string[] ; question: string }>`

Fetches a random question from the given theme pack, formatted for display.

**Parameters**

| Name | Type | Description |
|---|---|---|
| `theme_pack_name` | `string` | Name of the Theme Pack file in which a question is being generated |
| `category` | `string` | The name of the category that the question must belong to. |
| `question_type` | `"Multiple Choice" \| "Text Question"` | Denotes whether the question is multiple choice or text. |
| `used_questions` | `number[]` | A list of question ids in which have already been used by the game. |

**Returns** `Promise<{ id: number ; media_type: null | "image" | "video" ; media_url: null | string ; options: string[] ; question: string }>`

Formatted question data, loaded from file.

**Defined in** controllers/QuizController.ts:70

---

### validateAnswer

- **validateAnswer**(`themePackName`, `questionID`, `questionCategory`, `userAnswer`, `questionType?`): `Promise<boolean>`

Returns whether or not a user's answer to a question is correct.

**Parameters**

| Name | Type | Description |
|---|---|---|
| `themePackName` | `string` | Name of the Theme Pack file in which a question needs to be validated against. |
| `questionID` | `number` | The specific question id within that question file. |
| `questionCategory` | `string` | The category in which the question can be found in. |
| `userAnswer` | `string` | The user answer to the question, in which needs to be validated. |
| `questionType?` | `string` | The specific type of question asked, if known. |

**Returns** `Promise<boolean>`

**Defined in** controllers/QuizController.ts:25

## Module: models/Game

### Type Aliases

**GameType** T **GameType**: `Object`

The definition of what a game looks like within the database.

**Type declaration**

| Name | Type |
|---|---|
| game_code | string |
| hostId | mongoose.Types.ObjectId |
| players | mongoose.Types.ObjectId[] |
| started | boolean |
| theme_pack | string |
| used_consequences | number[] |
| used_questions | number[] |

**Defined in** models/Game.ts:12

### Variables

**default** • `Const` **default**: `Model<GameType, {}, {}, {}, any>`

**Defined in** models/Game.ts:45

## Module: models/User

### Type Aliases

**UserType** T **UserType**: `Object`

The definition of what a user looks like within the database.

**Type declaration**

| Name | Type |
|---|---|
| color | string |
| game | mongoose.Types.ObjectId |
| position | number |
| token | string |
| userType | string |
| username | string |

**Defined in** models/User.ts:12

### Variables

**default** • `Const` **default**: `Model<UserType, {}, {}, {}, any>`

**Defined in** models/User.ts:43

## Module: routes/basic.router

### Functions

**default**

- **default**(`instance`, `opts`, `done`): `void`

A universal router meant for handling requests that are non-node specific.

**Parameters**

| Name | Type |
| --- | --- |
| instance | FastifyInstance<RawServerDefault, IncomingMessage, ServerResponse<IncomingMessage>, FastifyBaseLogger |
| opts | Record<never, never> |
| done | (err?: Error) => void |

**Returns** void

**Defined in** node_modules/fastify/types/plugin.d.ts:13

## Module: routes/client.router

**Functions**

**default**

- **default**(instance, opts, done): void

The handling function for the client node router. It receives a request and various parameters, and handles it appropriately.

**Parameters**

| Name | Type |
| --- | --- |
| instance | FastifyInstance<RawServerDefault, IncomingMessage, ServerResponse<IncomingMessage>, FastifyBaseLogger |
| opts | Record<never, never> |
| done | (err?: Error) => void |

**Returns** void

**Defined in** node_modules/fastify/types/plugin.d.ts:13

## Module: routes/game.router

**Functions**

**default**

- **default**(instance, opts, done): void

The handling function for the game node router. It receives a request and various parameters, and handles it appropriately.

**Parameters**

| Name | Type |
| --- | --- |
| instance | FastifyInstance<RawServerDefault, IncomingMessage, ServerResponse<IncomingMessage>, FastifyBaseLogger |
| opts | Record<never, never> |
| done | (err?: Error) => void |

**Returns** void

**Defined in** node_modules/fastify/types/plugin.d.ts:13

## Module: routes/ws.router

**Functions**

**default**

- **default**(instance, opts, done): void

The handling function for the websocket router. It receives a request and various parameters, and handles it appropriately.

**Parameters**

| Name | Type |
| --- | --- |
| `instance` | `FastifyInstance<RawServerDefault, IncomingMessage, ServerResponse<IncomingMessage>, FastifyBaseLogger` |
| `opts` | `Record<never, never>` |
| `done` | `(err?: Error) => void` |

**Returns** `void`

**Defined in** node_modules/fastify/types/plugin.d.ts:13

## Setup and running

Please see section 7.1 Building and launching the project of our Software Design Document for details. This node will not function independently, so it is important to setup all the nodes before interacting with any them independently.