Backgammon Classic: Moves Calculation Logic

webtool.one

Source files

File name	Description
`main.js`	Initiation of `board` instance with respective parameters (see reference below).
`board.js`	Board class lives here. It contains logic for instantiating the board and producing the final output.
`piece.js`	The Piece class. Added to Board prototype.
`position.js`	The Position class. Added to Board prototype.
`logic.js`	The core for calculation. Includes main function `defMoves` and others. Added to Board prototype.
`logger.js`	Simple logger based on `console.log`. May be used as a starting point to include a full-featured logger.
`helpers.js`	Helper functions to perform actions on objects, arrays, etc.

Other files

File	Description
`calc.js`	File for demo. It may be used as an example of requiring calculation.
`test.js`	File for testing purposes.

Usage

In general, you need only to require `main.js` which provides a function for calculation. Then provide `gid`, `state`, and `params` to get a result. Please see for example `calc.js` (used for demo).

Basic example

```
/* Create function to get output */

const Board = require('./board')

const boardState = async (gid, state, params) => {
    const board = new Board(gid, state, params)
    await board.defMoves()
    return board.output
})

/* OR simply require main.js  */
const boardState = require('<path to main.js>')

/* Get actual state and send it to client */
const result = boardState(gid, state, params)

// ... do something with `result` ...
```

Position numbers

Note that position numbers here are used for calculation purposes only: the start is #1 and the end is #24. Blacks' drop-zone is #0 and whites' drop-zone is #25. If desired, these position numbers may be changed/converted after obtaining board output. If so, input to calculation should be re-converted too (pieces only).

Classes (prototypes)

Board

This is a core for calculation. The instance of the `Board` class is instantiated per request. It allows providing a state for each calculation making whole calculation process versatile. Also, due to the Board being stateless, it does calculations according to the provided state and parameters. For example, a state can be retrieved from a database in case of accidental disconnection.

In the normal way, the state should be seen as a snapshot of the board on client side and client-server communication can be as follows:

1. Client receives initial board state, i. e. all pieces are on respective heads. To obtain such initial state instantiate `Board` with no dice points;
2. A user does some moves: the modified state is sent for calculation;
3. The new `Board` instance is instantiated on the server with the actual `state` and calculations are performed;
4. A client receives an updated state with data on pieces and moves.

If no dice points are provided, heads will be set, i. e. all white pieces will be on position #1, and all black pieces will be on position #13 (for white color player view).
To get calculation results, use `Board.output` getter, which has the following properties:

Property	Type	Description
`activeUID`	`String | Number`	Used ID that has rights to do a move, i.e. user which has rolled dice.
`hasMovable`	`Boolean`	Designates whether there are any movable pieces.
`pieces`	See below	Pieces, i. e. board state, for each player. Positions are properly set according to each player's color.

Board output respects the view of each player and is suitable for immediate display. It means, for instance, that a white color player will see the 'home' of a black player as positions starting at #7 and a black color player will see his own 'home' starting at position #19.

`Board.output.pieces`
The structure of pieces exposed to output is as follows:

```
{
    '<user id #1>': { /* This is sutable for user #1 UI display. */
        self: Object{ '<piece ID>': Piece },
        opp: Object{ '<piece ID>': Piece } /* user #1 opponent's pieces */
    },

    '<user id #2>': { /* This is sutable for user #2 UI display. */
        self: Object{ '<piece ID>': Piece },
        opp: Object{ '<piece ID>': Piece } /* user #2 opponent's pieces */
    }
}
```

In `Board.output.pieces` object there are two representations of board layout, i.e. for each player. Each player piece layout respects the corresponding view: for a white color player the 'head' is on the top-right board section and the 'home' is on the bottom-right board section. At the same time, the black color player will see, for example, the white color player's 'home' at the top-left board section. So board layout is reversed for each player's view.

Piece

Class to work with pieces. The `Piece` instance is exposed in board output, so can be used for UI purposes. It has the following properties exposed to `Board.output`:

Property	Type	Description
`gid`	`String | Number`	ID of game.
`uid`	`Number`	ID of the user this piece belongs to.
`pid`	`Number`	ID of this piece. Useful for positioning within client UI.
`color`	`Number`	Color of piece: 0 for white and 1 for black.
`posN`	`Number`	Actual piece position.
`iPosN`	`Number`	Initial piece position. As a player can do more than one move after the dice roll, this property keeps the starting point of movements and is used to undo moves.
`index`	`Number`	As on one position can be several pieces, there can be a 'column' of pieces. The `index` shows the exact place of this particular piece concerning the order in 'column'. It's also is used to undo moves: the piece's place will be reverted to the highest position in the 'column'.
`posToMove`	`Array<Number>`	Array of position numbers this piece can be moved to. It includes piece's current position number: user may take and move piece around but finally place it at initial position (actual move has not been done).
`movements`	`Array<Number>`	Position numbers this piece has been moved to on previous steps.
`movable`	`Boolean`	Whether this piece is movable. In case there are several pieces on one position, only piece on top of such 'column' is movable.
`droppedOut`	`Boolean`	Whether this piece is dropped out.

Position

Class for representing a particular board position. Used for internal purposes only. For more information see source: `position.js`.

Parameters for calculation

These parameters are passed to the main function (which initializes `Board` and returns output, i. e. `main.js`) to produce a result. For usage examples please see `calc.js` and `test.js`. A general description is provided below.

`gid`
Type: `String | Number`

ID of in-progress game. Can be used to identify the current game and/or retrieve data from database to restore the game state.

`state`
Type: `Object`

Includes all information about actual game state which will be used for calculation.

Property	Type	Description
`pieces`	`Array<Piece>`	Array of pieces with actual position, done moves, etc.
`colors`	`Object{ <user id>: <color: 0 | 1> }`	Colors of each player mapped to their IDs: 0 = white, 1 = black, e. g. `{ 1000: 0, 1001: 1 }`,
`dicePoints`	`Array<Number>`	Dice points, e. g. `[2, 5]`. If no dice points are provided, heads will be set. Dice points should be passed 'as is', because done moves are retrieved from every `Piece` (`Piece.movements`).
`status`	`String`	Status of game, e. g. 'in-progress'. Optional, not in use. May be useful for tracking game progress.
`activeUID`	`Number`	User ID moves are calculated for.

`params`
Type: `Object`

All parameters are optional and defaults to `false`.

Property	Type	Description
`undo`	`Boolean`	Whether this is an undo action: pieces will be reverted to their initial positions.
`fixed`	`Boolean`	Discard all previous movements, i. e. each piece current position will be set as initial.
`outOnly`	`Boolean`	Don't do any calculations (even don't discard movements), return pieces 'as is'. Useful to restore game state.

Please see the source files for additional information.

Web version

Demo