That's a great request! Moving this implementation to a production-ready, scalable web service requires formalizing the flow, defining clean APIs, and establishing a persistent data schema.

🚀 Game Flow Explanation

The provided Java implementation follows a clear, layered sequence to execute the game from setup to completion.

1. Setup Phase

Step	Method	Description
1. Initialize Game	GameManager.initGame(N) $\rightarrow$ GameService.initGame(N)	A unique **Game** object is created with an ID, the battlefield size $N \times N$, and the default **RandomCoordinateFireStrategy**. The game is registered in the GameManager's active map to handle concurrent games.
2. Add Ships	GameManager.addShip(...) $\rightarrow$ GameService.addShip(...)	**Input validation** checks if the ship's size and center coordinates keep the entire ship within the player's designated half and prevent overlaps. The ship's coordinates are calculated and stored in the **Game.battlefieldGrid** map (key: Coordinate, value: Ship ID). Equal fleets are assigned to Player A and Player B.
3. View Battlefield	GameManager.viewBattleField(...) $\rightarrow$ GameService.viewBattleFiel	This uses the Game.battlefieldGrid to render the current state of the board, showing where

	d(...)	all ships are located.

---

2. Execution Phase (The Game Loop)

The game begins with a call to GameManager.startGame(...), which enters the main game loop (while (!game.isGameOver())).

Step	Player	Method/Logic	Outcome
1. Player Turn	Player A (first) $\rightarrow$ Player B $\rightarrow$ A...	GameService.performTurn(game, currentPlayer, opponentPlayer)	The turn starts, identifying the opponent's field boundaries (e.g., Player A targets $X \in [N/2, N)$).
2. Fire Missile	Current Player	FireStrategy.getNextTarget(...)	The **Strategy Pattern** is used to determine a **new, unique target** Coordinate in the opponent's field. The Game.firedCoordinates set ensures no coordinate is targeted twice.
3. Result Check	Current Player	Logic checks the target Coordinate against the opponent's ships (stored in Game.battlefieldGrid).	**"Hit"**: If a ship is found, it is marked as isDestroyed = true. **"Miss"**: If no ship is found.
4. State Update	Both	Game.playerB.getRemainingShipsCount() is checked.	The remaining ships count is printed.

5. End Check	Both	If the opponent's remaining ship count is 0.	**Game Over!** The current player is set as the winner.
6. Switch Turn	Both	Player A becomes Player B, and vice versa.	The loop continues until the game is over.

🏗️ Low-Level Design for Production (APIs and Persistence)

For a scalable web application, we must transition the in-memory models to a persistent database (e.g., PostgreSQL, MySQL) and define RESTful APIs.

1. Production APIs

These APIs would be exposed via a REST controller (e.g., using Spring Boot or similar framework), with the GameManager logic residing in a service layer.

API Endpoint	HTTP Method	Functionality	Request Body (Input)	Success Response (Output)
/games	POST	**1. Create New Game**	{"N": 6}	{"gameId": "UUID-123", "N": 6, "status": "SETUP"}
/games/{gameId}	GET	**2. Get Game State**	*(None)*	{"gameId": "UUID-123", "status": "IN_PROGRESS", "board": ..., "nextTurn": "PlayerA", ...}
/games/{gameI	POST	**3. Add Ship to**	{"id": "SH1", "size": 2, "xA":	{"message": "Ship SH1

d}/ships		**Game**	1, "yA": 5, "xB": 4, "yB": 4}	added successfully to both fleets."}
/games/{gameId}/start	POST	**4. Start Game**	*(None)*	{"message": "Game started. PlayerA's turn.", "nextTurn": "PlayerA"}
/games/{gameId}/fire	POST	**5. Execute Turn/Fire Missile**	{"playerId": "PlayerA"} *(The coordinates are determined by the server's strategy)*	{"result": "Hit", "coordinate": "(5, 3)", "destroyedShip": "B-SH2", "shipsRemainingA": 1, "shipsRemainingB": 0, "status": "GAME_OVER", "winner": "PlayerA"}
/games/{gameId}/board	GET	**6. View Battlefield** (Optional)	*(None)*	{"grid": [["A-SH1", ...], ...]} *(Detailed grid state)*

---

2. Database Tables (Persistence Layer)

To maintain state between server restarts and support the web environment, we need to persist the game, players, and ships.

Table: GAME

Stores the high-level game information.

Column Name	Data Type	Constraint	Description

game_id	VARCHAR(36)	**Primary Key**	Unique ID for the game.
N	INTEGER	NOT NULL	Battlefield size (e.g., 6).
status	VARCHAR(20)	NOT NULL	SETUP, IN_PROGRESS, GAME_OVER.
turn	VARCHAR(10)	NOT NULL	PLAYER_A or PLAYER_B.
winner	VARCHAR(10)	NULLABLE	PLAYER_A or PLAYER_B if GAME_OVER.
fire_strategy	VARCHAR(50)	NOT NULL	Class name of the strategy (e.g., RandomCoordinate FireStrategy).

Table: SHIP

Stores details for every ship instance in the game.

Column Name	**Data Type**	**Constraint**	**Description**
ship_id	BIGINT	**Primary Key** (Auto-increment)	Unique ID for the specific ship instance.
game_id	VARCHAR(36)	**Foreign Key** (GAME)	The game this ship belongs to.
player_name	VARCHAR(10)	NOT NULL	PLAYER_A or PLAYER_B.

ship_code	VARCHAR(10)	NOT NULL	The user-defined ID (e.g., "SH1", "SH2").
size	INTEGER	NOT NULL	Ship size.
is_destroyed	BOOLEAN	NOT NULL	True if the ship has been hit and destroyed.

Table: BATTLEFIELD_COORDINATE

This is the most critical table, mapping every coordinate to its state: occupied ship, or fired status. It represents the persistent state of the $N \times N$ grid.

Column Name	Data Type	Constraint	Description
coord_id	BIGINT	**Primary Key** (Auto-increment)	Unique ID for the grid cell state.
game_id	VARCHAR(36)	**Foreign Key** (GAME)	The game this coordinate belongs to.
x_coord	INTEGER	NOT NULL	X-coordinate (0 to N-1).
y_coord	INTEGER	NOT NULL	Y-coordinate (0 to N-1).
occupied_by_ship_id	BIGINT	**Foreign Key** (SHIP)	The ship_id if a ship occupies this cell. Null otherwise.
is_fired	BOOLEAN	NOT NULL	True if a missile has been fired at this coordinate.

Would you like to explore how to implement a different firing strategy, such as one that targets

an adjacent cell after a miss?