# Network Protocol

The following are requests from the client to the server and expected responses from the server to the client. It is essential here that most responses are encapsulated by a general ResponseEnvelope. This records the actual data that is to be transmitted and also offers the option of communicating whether a request from the client was correct or incorrect.

In addition, an error message is typically returned in the event of an incorrect request. Another special case is the initialization of a new game. To do this, a specified endpoint is accessed using an HTTP GET operation and the server responds accordingly with a game ID. This never fails.

## 1. Creation of a new game

Before two clients can compete against each other, a new game must be created on the server. For this it is necessary to access the endpoint described below by sending an HTTP GET request to it. A normal web browser is all that is needed for this.

Client Header: Since the response is defined with data in XML format, the request should include an appropriate "accept" HTTP header, hence "accept = application/xml".

**Client request**

Send an HTTP GET operation to the following endpoint **http://<domain>:<port>/games**
The server responds with an XML message containing a unique GameID. Details on this can be found below. Since this game ID is unique for each game, it can be used in the following to assign messages to a specific game and thus the server can distinguish for which game an incoming message is intended. This allows multiple games to be played in parallel by different players.

**Server response**

This response is sent in response to the new game creation request described above. The main part of the message defined in XML contains an element called uniqueGameID, which contains a unique game ID (this ID must consist of exactly 5 characters). This GameID must be used as part of the endpoint address in the following requests, so that the server knows which game a message is intended for. Creating a new game is always possible and must never fail.

**XML schema of the message**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="uniqueGameIdentifier">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="uniqueGameID" maxOccurs="1" minOccurs="1">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:minLength value="5" />
              <xs:maxLength value="5" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
```

```
  </xs:element>
</xs:schema>
```

**Message Example**: *A game's unique GameID would be "Ss8Zc" in the example below.*

```
<?xml version="1.0" encoding="UTF-8"?>
<uniqueGameIdentifier>
  <uniqueGameID>Ss8Zc</uniqueGameID>
</uniqueGameIdentifier>
```

## 2. Registration of a client

As soon as a game has been created and both clients know the unique game ID (see above), the clients can register for the new game. It is important that no mechanism is provided to communicate the same game ID to both clients directly from the server. It is therefore intended that the game ID is generated once by a person (e.g. by accessing the endpoint via a web browser to create a new game). This allows both clients to start interacting with the server at about the same time.

**Client request**

The client sends an HTTP POST request with an XML message in the body to the following endpoint: **http://<domain>:<port>/games/<GameID>/players**

Here, the <GameID> part is to be replaced with the unique GameID obtained during game creation. Information about the client is transmitted as the body of the message. This would be the username (playerUsername).

**XML schema of the message**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="playerRegistration">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="playerUsername" maxOccurs="1" minOccurs="1">
               <xs:simpleType>
                  <xs:restriction base="xs:string">
                     <xs:minLength value="1" />
                     <xs:maxLength value="50" />
                  </xs:restriction>
               </xs:simpleType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

**Message Example**

```
<?xml version="1.0" encoding="UTF-8"?>
<playerRegistration>
  <playerUsername>john_doe</playerUsername>
```

</playerRegistration>

**Server response**
The server responds to the registration message (see above) with a confirmation that the registration has been carried out and a corresponding unique player ID. This player ID must be transmitted with the following requests so that the server knows from which client (or player) this request was made.

This and the following messages from the server are enclosed in a responseEnvelope element. This responseEnvelope is designed for two cases, which can be distinguished via the state element. If the state element contains the text Okay, the request could be processed correctly and the expected data (like in this case the player ID) can be found in the data element. However, if the state element shows the value Error, then an error has occurred, e.g. because a game rule has been violated. In this case, the name of an associated error message can be found in the exceptionName element and a note about the error that has occurred in the exceptionMessage element. This behavior is also applied to all incoming messages and helps you to identify errors in the client.

**XML schema of the message**
```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="responseEnvelope">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="exceptionName" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="exceptionMessage" type="xs:string" minOccurs="1" maxOccurs="1"
/>
        <xs:element name="state" type="statevalues" minOccurs="1" maxOccurs="1" />
        <xs:element name="data" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="uniquePlayerIdentifier">
    <xs:sequence>
      <xs:element name="uniquePlayerID" type="xs:string" minOccurs="1" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="statevalues">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Okay" />
      <xs:enumeration value="Error" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

**Message Example**
```
<?xml version="1.0" encoding="UTF-8"?>
<responseEnvelope>
  <exceptionName/>
```

```
  <exception message/>
  <state>Okay</state> <data xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="uniquePlayerIdentifier">
  <uniquePlayerID>702ecdb6-b363-462c-a63a-5d0a39222c4e</uniquePlayerID>
 </data>
</responseEnvelope>
```

## 3. Game status query

The query of a game status by the client can only begin after the registration of a client since the game ID as well as the player ID are required for this. When querying the game status, the server returns all the game details that are relevant for the clients. This is the full map and information to coordinate clients' actions (whether a client should take an action or whether a client won or lost).

To prevent the server from being overloaded, there must be at least 0.4 seconds between two game status queries made by the same client. The client must ensure this. A simple/fast, albeit not pretty, implementation for this is Thread.sleep between two game state queries.

**Client request**
The client sends an HTTP GET request to the following endpoint:
**http(s)://<domain>:<port>/games/<GameID>/states/<PlayerID>**

In this case, the part GameID is to be replaced with the unique GameID received during game generation and the PlayerID with the unique PlayerID communicated by the server during registration. If both IDs are known to the server, the server responds with the game status, otherwise an appropriate error message is returned.

Warning: Since the response is defined with data in XML format, the request should include an appropriate "accept" HTTP header, hence "accept = application/xml".

**Server response**
The server replies with a responseEnvelope which, in the event of an error (as with the previous messages), contains a corresponding error description and, with the state element, clarifies whether an error has occurred or not. The data element contains the actual user data. Hence, basically the information about the players and the map. For a player, uniquePlayerID, playerUsername, state (status of the player) are appended. The uniquePlayerID corresponds to the unique PlayerID created and sent during registration. The latter only applies to the data record that represents the requesting player, the player IDs of the other players do not correspond to their real player IDs. The state in turn shows the status of the client, which can either be that the server is expecting a new command for this client (MustAct), that no command is expected (MustWait), that the client has lost (Lost) or that the client has won (Won).

Above this is the game map in the map element. For each map field it is displayed whether there are avatars of the clients on the field (playerPositionState), what terrain the field has (terrain), whether there is a treasure on the field (treasureState) and whether there is a fort on the field (fortState). Furthermore, the coordinates are given, therefore X coordinate and Y coordinate.

The playerPositionState element indicates whether there is no avatar on the field (NoPlayerPresent), the enemy client's avatar is on the field (EnemyPlayerPosition), the requesting client's avatar is on the field (MyPlayerPosition), or avatars of both clients (BothPlayerPosition ). The terrain element indicates the field type as Water, Grass or Mountain. With regard to the treasure (treasureState) it can either be reported that there is no treasure on it or it is not known whether there is a treasure on the field (NoOrUnknownTreasureState). Alternatively, it is communicated that one's own treasure is on the field (MyTreasureIsPresent). Once a treasure has been "picked up", meaning the appropriate avatar was on the space where a treasure was reported, MyTreasureIsPresent is no longer reported as true. With regard to the castle, the element fortState either has the value NoOrUnknownFortState if it is not known whether there is a castle on the field or there is no castle there, or alternatively it is indicated that your own castle is there (MyFortPresent) or the enemy's (EnemyFortPresent ). Once a castle has been uncovered, it is noted on the map in each subsequent query (this also applies to treasures that have not been picked up but have been uncovered).

The gameStateId element is also relevant. This contains a randomly created ID, which makes it possible to recognize whether the status information supplied by the server has changed since the last query. If this is the case, the gameStateId has changed. Furthermore, the collectedTreasure flag in the player element shows whether the respective client has already found the treasure or not. This information, in turn, can be used to determine whether the client is still looking for the treasure or is already looking for the opposing castle.

Information on the ordering/sorting of the content: The player details and the card details (fields) are returned from the server in an unordered manner. Therefore, use the PlayerID (for the player) and the X/Y coordinates (for the map or fields) to query or determine the appropriate information for you.

**XML schema of the message**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="responseEnvelope">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="exceptionName" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="exceptionMessage" type="xs:string" minOccurs="1" maxOccurs="1"
/>
        <xs:element name="state" type="statevalues" minOccurs="1" maxOccurs="1" />
        <xs:element name="data" minOccurs="0" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
    <xs:complexType name="gameState" >
      <xs:sequence>
        <xs:element ref="players" minOccurs="1" maxOccurs="1" />
        <xs:element ref="map" minOccurs="0" maxOccurs="1" />
        <xs:element ref="gameStateId" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
```

```
  <xs:element name="players">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="2" minOccurs="1" ref="player" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="player">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="uniquePlayerID" minOccurs="1" maxOccurs="1" />
        <xs:element ref="playerUsername" minOccurs="1" maxOccurs="1" />
        <xs:element name="state" type="playerGameStatevalues" minOccurs="1"
maxOccurs="1" />
        <xs:element ref="collectedTreasure" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="uniquePlayerID" type="xs:string" />
  <xs:element name="playerUsername" type="xs:string" />
  <xs:element name="collectedTreasure" type="xs:boolean" />
  <xs:element name="map" minOccurs="1" maxOccurs="1">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="mapNodes" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="mapNodes">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="100" maxOccurs="100" ref="mapNode" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="mapNode">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="1" ref="playerPositionState" />
        <xs:element minOccurs="1" maxOccurs="1" ref="terrain" />
        <xs:element minOccurs="1" maxOccurs="1" ref="treasureState" />
        <xs:element minOccurs="1" maxOccurs="1" ref="fortState" />
        <xs:element minOccurs="1" maxOccurs="1" ref="X" />
        <xs:element minOccurs="1" maxOccurs="1" ref="Y" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="playerPositionState" type="playerStatevalues" />
  <xs:element name="terrain" type="terrainStatevalues" />
```

```
<xs:element name="treasureState" type="treasureStatevalues" />
<xs:element name="fortState" type="fortStatevalues" />
<xs:element name="X">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0" />
      <xs:maxInclusive value="19" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="Y">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0" />
      <xs:maxInclusive value="9" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="gameStateId" type="xs:string" />
<xs:simpleType name="statevalues">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Okay" />
    <xs:enumeration value="Error" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="playerStatevalues">
  <xs:restriction base="xs:string">
    <xs:enumeration value="NoPlayerPresent" />
    <xs:enumeration value="EnemyPlayerPosition" />
    <xs:enumeration value="MyPlayerPosition" />
    <xs:enumeration value="BothPlayerPosition" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="terrainStatevalues"> </xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:enumeration value="Water" />
    <xs:enumeration value="Grass" />
    <xs:enumeration value="Mountain" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="playerGameStatevalues">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Won" />
    <xs:enumeration value="Lost" />
    <xs:enumeration value="MustAct" />
    <xs:enumeration value="MustWait" />
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="treasureStatevalues">
```

```
    <xs:restriction base="xs:string">
      <xs:enumeration value="NoOrUnknownTreasureState" />
      <xs:enumeration value="MyTreasureIsPresent" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="fortStatevalues">
    <xs:restriction base="xs:string">
      <xs:enumeration value="NoOrUnknownFortState" />
      <xs:enumeration value="MyFortPresent" />
      <xs:enumeration value="EnemyFortPresent" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

**Message Example**

```
<?xml version="1.0" encoding="UTF-8"?>
<responseEnvelope>
  <exception name />
  <exception message />
  <state>Okay</state>
  <data xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="gameState">
    <players>
      <player>
        <uniquePlayerID>6c4362b5-5636-41b8-aaae-6daba4aea91b</uniquePlayerID>
        <playerUsername>john_doe</playerUsername >
        <state>MustAct</state>
        <collectedTreasure>false</collectedTreasure>
      </player>
      <player>
        <uniquePlayerID>2e31d1c8-e91d-4f55-88ad-ffcd1dae48f0</uniquePlayerID>
        <playerUsername >max_example</playerUsername >
        <state>MustWait</state>
        <collectedTreasure>false</collectedTreasure>
      </player>
    </players>
    <map>
      <mapNodes>
        <mapNode>
          <playerPositionState>NoPlayerPresent</playerPositionState>
          <terrain>Water</terrain>
          <treasureState>NoOrUnknownTreasureState</treasureState>
          <fortState>NoOrUnknownFortState</fortState>
          <X>2</X>
          <Y>1</Y>
        </mapNode>
        <mapNode>
          <playerPositionState>NoPlayerPresent</playerPositionState>
          <terrain>Grass</terrain>
          <treasureState>NoOrUnknownTreasureState</treasureState>
```

```
          <fortState>NoOrUnknownFortState</fortState>
          <X>6</X>
          <Y>4</Y>
        </mapNode>
      </mapNodes>
    </map>
    <gameStateId>430bdf29-5e61-4950-8408-ac40e9fb1176</gameStateId>
  </data>
</responseEnvelope>
```

*Shortened example: In order to keep the example short, not all mapNode elements were listed.*

## 4. Transmission of a movement

When it is a client's turn (hence the game status shows that the next action needs to be sent), the client can broadcast a game move. Movements are always transmitted step by step, so moving between two grass fields, for example, requires 2 movements in the direction (a "movement message" from a client always transmits only one movement in one direction of movement) of the corresponding field in order to leave the old grass field and to enter the new one to enter.

**Client request**

The client sends an HTTP POST request to the following endpoint:
**http(s)://<domain>:<port>/games/<GameID>/moves**

The request contains XML data in the body and is also answered with XML data. Therefore, the request should include an appropriate "accept" HTTP header, hence "accept = application/xml". The data that is transmitted in the body consists of the player ID, which was communicated to the client when the player was registered (uniquePlayerID) and the movement command (move). The movement command contains the direction in which the avatar, which is controlled by the client, should move. Therefore the avatar can move up (Up), left (Left), right (Right), as well as down (Down).

**XML schema of the message**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="playerMove">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="uniquePlayerID" minOccurs="1" maxOccurs="1" />
        <xs:element ref="move" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="uniquePlayerID" type="xs:string" />
  <xs:element name="move" type="moveValues" />
  <xs:simpleType name="moveValues">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Up" />
```

```
      <xs:enumeration value="Down" />
      <xs:enumeration value="Left" />
      <xs:enumeration value="Right" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

**Message Example**

```
<?xml version="1.0" encoding="UTF-8"?>
<playerMove>
  <uniquePlayerID>1c44419e-304a-4caa-8901-b7d8c62898b5</uniquePlayerID>
  <move>Right</move>
</playerMove>
```

**Server response**

The server responds generically with a responseEnvelope. However, since no data is generated here, the data element is not included in this case. Apart from that, the behavior is as in the message described earlier. Therefore, in the event of a problem, the exceptionName and exceptionMessage are filled with details about the problem and the state element is filled with the text Error. If there is no error, state is filled with the value Okay.

**XML schema of the message**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="responseEnvelope">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="exceptionName" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element name="exceptionMessage" type="xs:string" minOccurs="1" maxOccurs="1"
/>
        <xs:element name="state" type="statevalues" minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="statevalues">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Okay" />
      <xs:enumeration value="Error" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

**Message Example**

```
<?xml version="1.0" encoding="UTF-8"?>
<responseEnvelope>
  <exception name />
  <exception message />
  <state>Okay</state>
</responseEnvelope>
```