TH-L WiSe 23/24: Patterns & Frameworks

Worms-like Scampi di Mare

Schnittstellenbeschreibung

Autoren: GruppeL

Timo Nadolny
Lennart Sparbier
Alexander Voigt

Inhalt

Controller Beschreibung im Javadoc Stil

AuthenticationController

```java
/***
 * Controller für die Nutzerauthentifizierung sowie Registrierung neuer
 * Nutzer. Als Basisverzeichnis wird /authenticate genutzt.
 */
@Controller
@RequestMapping("authenticate")
public class ShrimpServerAuthenticationController {

    /***
     * Zugriff auf das Verzeichnis aller Spieler
     */
    @Autowired
    public PlayerRepository repo;

    /***
     * Stellt einen Funktionstest für den Authentifikationscontroller
     * bereit.
     * @return String Meldung, dass der Test erfolgreich war.
     */
    @ResponseBody
    @GetMapping("/test")
    public String testingController() {…}

    /***
     * Gibt ein Formular zurück, mit dem sich ein neuer Spieler
     * registrieren kann.
     * @param model Es wird eine Referenz auf ein neues
     *              Spieler-Objekt mitgegeben.
     * @return  Der Verweis auf das Formular-Template.
     */
    @RequestMapping(value = "/register", method = RequestMethod.GET)
    public String viewRegistrationForm(Model model) {…}

    /***
     * Registriert einen neuen Spieler in der Datenbank. Zuvor wird
     * das übergebene Passwort verschlüsselt. Dies sollte bei einem
     * Produktivsystem bereits im jeweiligen Client passieren.
     *
     * @param player    Das Spieler-Objekt, das hinzufügt werden soll.
     * @return  Verweis auf die Seite für erfolgreiche Registrierung
     */
    @PostMapping("/process_register")
    public String processRegister(Player player) {…}
```

LobbyController

```java
/***
 * Der LobbyController stellt die Dienste des LobbyService
 * per RestAPI zur Verfügung unter dem Präfix "/lobby". Die
 * Dienste umfassen hierbei das Senden und Empfangen von
 * Nachrichten, direkte sowie allgemeine. Neu eintreffende
 * Nachrichten werden hierbei ebenso über einen WebSocket
 * bekannt gegeben.
 */
@Controller
public class ShrimpServerLobbyController {

    /***
     * Service für die angebotenen Dienste
     */
    @Autowired
    public LobbyService lobbyService;

    /***
     * Service für spielerbezogene Dienste
     */
    @Autowired
    public PlayerDetailsService playerService;

    /***
     * Initialisiert mit einer neuen Service-Instanz
     */
    ShrimpServerLobbyController() {…}

    /***
     * Initialisiert mit übergebener Service-Instanz
     *
     * @param service   Die übergebene Service-Instanz
     */
    ShrimpServerLobbyController(LobbyService service) {
        this.lobbyService = service;
    }

    /***
     * Zeigt die Lobby Seite an
     *
     * @param model    Daten für das Binding an die Seite
     * @return         Die Lobby Seite
     */
    @GetMapping("/lobby")
    public String viewLobby(Model model, Principal principal) {…}

    /***
     * Gibt die persönlichen Nachrichten des eingeloggten Users aus
     *
     * @param principal        Der eingeloggte Nutzer
     * @return                 Liste von Nachrichten
     */
    @GetMapping("/lobby/get_private_messages/")
    public List<Message> getPrivateMessages(Principal principal) {…}
```

```java
/***
 * Ermöglicht das Versenden von Nachrichten. Sender muss dabei
 * zwingend gefüllt sein. Durch die gesetzten Attribute wird die
 * Natur der Nachricht bestimmt.
 * Fall 1, Sender und Empfänger != null, GameInvite == null:
 * Es handelt sich um eine direkte Nachricht
 * Fall 2, Sender, Empfänger und GameInvite != null:
 * Eine Einladung in eine Partie
 * Fall 3, Sender != null, Empfänger und GameInvite == null:
 * Eine Nachricht für die Lobby
 * Fall 4, Sender != null, Empfänger = null und GameInvite != null:
 * Nachricht für Partie-Chat
 *
 * @param message   Die Nachricht die übermittelt wird
 * @return  String  Ok falls valide, sonst Err
 */
@PostMapping("/lobby/send_message")
public String sendMessage(Message message) {…}

/***
 * Gibt die 20 neusten Einträge im Lobby-Chat zurück
 *
 * @return  List of Message, Die Nachrichtenliste
 */
@GetMapping("/lobby/top20LobbyChat")
@ResponseBody
public List<Message> getTop20LobbyChat() {… }

/***
 * Gibt alle Nachrichten der Lobby zurück, sortiert von neuster
 * bis zur ältesten Nachricht.
 *
 * @return  Liste der Nachrichten
 */
@GetMapping("/lobby/getAllLobbyChat")
@ResponseBody
public List<Message> getLobbyChat() {…}

/***
 * Gibt eine Liste aller, zurzeit in der Lobby eingeloggten Spieler
 * zurück.
 *
 * @return  Liste der Spieler
 */
@GetMapping("/lobby/onlineUsers")
@ResponseBody
public List<Player> getOnlineUsers() {…}

/***
 * WebSocket Schnittstelle
 * Nachrichten die an /shrimps/socket.sendMessage gesandt werden,
 * werden direkt auf /chat/Messages ausgegeben.
 *
 * @return Liste aktueller Nachrichten
 */
@MessageMapping("/socket.sendMessage")
@SendTo("/chat/messages")
public Message sendSocketMessage(@Payload Message message) {…}
```

```
    /***
     * Registrierungseinstieg für WebSocket Verbindungen für die Lobby.
     * Der Spieler wird der Lobby hinzugefügt, der Username wird der
     * Session hinzugefügt, eine Meldung zum Einloggen in der Lobby wird
     * gesendet, sowie der Online-Status in der Datenbank erfasst.
     *
     * @param message           Die Meldung in den Lobby-Chat (incoming)
     * @param headerAccessor    Die Sessionvariable
     * @return                  Nachricht für die Lobby (outgoing)
     */
    @MessageMapping("/socket.addUser")
    @SendTo("/chat/messages")
    public Message addUser(@Payload Message message,
                           SimpMessageHeaderAccessor headerAccessor) {…}
}
```

GameController

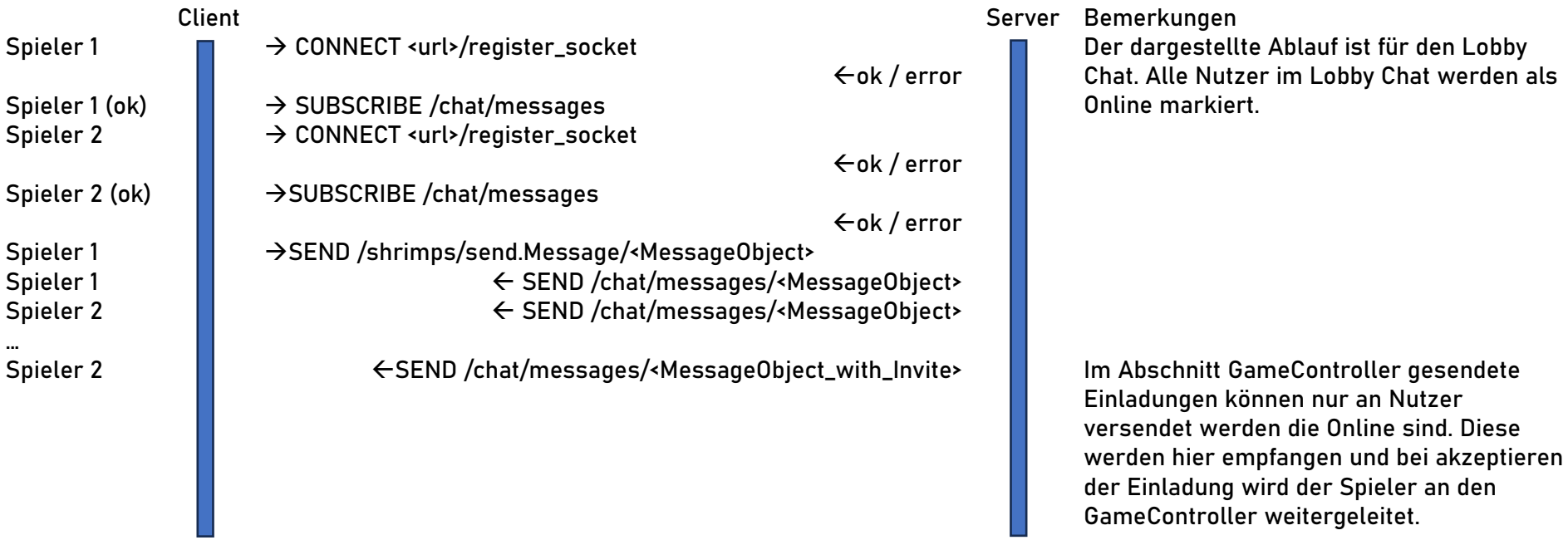
```java
/***
 * Controller für die Bereitstellung der auf das Spiel bezogenen
 * Dienste. Während die Initialisierung des Spiels, wie das Finden
 * von Spielern sowie die initiale Konfiguration werden hierbei per
 * REST API durchgeführt, während des Spiels findet die Client - Server
 * Kommunikation per WebSocket statt.
 */
@Controller
@RequestMapping("/game")
public class ShrimpServerGameController {

    /***
     * Service für bereitgestellte Dienste
     */
    @Autowired
    ShrimpGameService gameService;

    /***
     * Service für Spieler-basierte Dienste
     */
    @Autowired
    PlayerRepository playerRepository;

    /***
     * Diese REST Schnittstelle bietet für den integrierten Spring
     * Prototyp-Client die Konfigurationsseite, um eine Spielpartie zu
     * konfigurieren. Hierbei kann nur der "Hostende"-Spieler
     * Spielgrundeinstellungen ändern [game.hostingPlayer]. Die
     * anderen Spieler können die Namen Ihrer TeamShrimps ändern.
     *
     * @param model         Model um Attribute für Thymeleaf zu "injecten"
     * @param principal     Der Aufrufer dieser Methode
     * @return              Die Konfigurationsseite des Spiels
     */
    @RequestMapping(value = "/configureSpring",
                    method = RequestMethod.POST)
    public String configureGame(Model model,
                                Principal principal) {…}

    /***
     * REST Schnittstelle für die Spielseite für den integrierten Spring-
     * Thymeleaf-Client. Auf dieser Seite findet das Spiel statt bis die
     * Partie beendet wurde.
     *
     * @param gameId   Die Spiel Id um die Partie zu identifizieren
     * @param model           Modell der Seite für Thymeleaf
     * @param principal       Der Aufrufer der Seite
     * @return                Die HTML-Thymeleaf-Seite
     */
    @RequestMapping(value = "/{gameId}/playSpring",
                    method = RequestMethod.POST)
    public String playGame(@PathVariable Long gameId,
                           Model model,
                           Principal principal) {…}
```

```java
    /***
     * Diese REST-API Methode wird vom hostenden Spieler aufgerufen, um das
     * Spiel zu beginnen. Dies ist nur erforderlich, sofern es sich nicht
     * um den integrierten Spring-Client handelt.
     *
     * @param gameId        Die Spiel Id um die Partie zu identifizieren
     * @param principal     Der Aufrufer des Services zur validierung der
                            Berechtigung
     */
    @ResponseBody
    @RequestMapping(value = "/{gameId}/play", method = RequestMethod.POST)
    public void startGame(@PathVariable Long gameId,
                                        Principal principal) {…}

    /***
     * REST Schnittstelle um ein Spiel zu erstellen, die neu erstellte
     * Partie wird dann zurückgegeben. Änderungen werden per
     * /{gameId}/changeGame eingestellt. Diese Schnittstelle wird nicht vom
     * integrierten Client bedient.
     *
     * @return Game-Objekt inklusive der entsprechenden Id die für die
     *         Socket-Pfadangabe benötigt wird
     */
    @ResponseBody
    @RequestMapping(value = "/startNewGame", method = RequestMethod.POST)
    public Game getGameId() {…}

    /***
     * WebSocket - Schnittstelle um Änderungen der Konfiguration
     * vorzunehmen. Die Änderungen werden anschließend über diese
     * Schnittstelle an alle Abonnenten verteilt.
     *
     * @param gameId        Die Spiel Id um die Partie zu identifizieren
     * @param principal     Der Aufrufer zur Prüfung der Berechtigung
     * @param game          Das Game-Objekt um Änderungen zu verteilen
     * @return              Das veränderte, validierte Game-Objekt
     */
    @MessageMapping("/{gameId}/changeGame")
    @SendTo("/{gameId}/gameChanged")
    private Game changeGame (@DestinationVariable Long gameId,
                                    Principal principal,
                                    Game game) {…}

    /***
     * WebSocket - Schnittstelle um eine Bewegung einer Spielfigur an den
     * Server zu melden. Diese wird überprüft und sofern valid an die
     * Clients verteilt.
     *
     * @param gameId        Die Spiel Id um die Partie zu identifizieren
     * @param principal     Der Aufrufer zur Prüfung der Berechtigung
     * @param playerMove    Die gewünschte Bewegung
     * @return              Die Bewegung die vom Server genehmigt wurde
     */
    @MessageMapping("/{gameId}/makeMove")
    @SendTo("/{gameId}/moveMade")
    public PlayerMovedDTO makePlayerMove(@DestinationVariable Long gameId,
                                        Principal principal,
                                        PlayerMoveDTO playerMove) {…}
```

```java
    /***
     * WebSocket - Schnittstelle um das Abfeuern einer Waffe einer
     * Spielfigur durchzuführen. Die Anforderung wird vom Server validiert
     * und eine entsprechende Antwort wird an die beteiligten Clients
     * verteilt. Sollte der Waffeneinsatz eine Änderung an der
     * Spiellandschaft verursachen, wird dies über den entsprechenden
     * Socket /{gameId}/gameChange an die Clients verteilt. Shrimps die
     * durch eine entsprechende Lücke fallen werden über /{gameId}/moveMade
     * verteilt.
     *
     * @param gameId        Die Spiel Id um die Partie zu identifizieren
     * @param principal     Der Aufrufer zur Prüfung der Berechtigung
     * @param fireWeapon    Die Information über den gewünschten
     *                      Waffeneinsatz.
     * @return              Information über die Flugbahn des Geschosses,
     *                      den angerichteten Schaden, u.ä.
     */
    @MessageMapping("/{gameId}/fireWeapon")
    @SendTo("/{gameId}/weaponFired")
    public FiredWeaponDTO fireWeapon(@DestinationVariable Long gameId,
                                     Principal principal,
                                     FireWeaponDTO fireWeapon) {…}

    /***
     * WebSocket - Schnittstelle über den die Clients melden, dass sie die
     * Verarbeitung der aktuellen Meldungen abgeschlossen haben. Sobald
     * alle Spieler der Partie gemeldet haben, dass sie bereit sind, fährt
     * der Server damit fort über /{gameId}/gameChanged zu melden welcher
     * Spieler nun an der Reihe ist.
     *
     * @param gameId        Die Spiel Id um die Partie zu identifizieren
     * @param principal     Der Aufrufer des Services zur identifikation
     *                      wer bereit ist
     * @return              Systemmeldung an die Clients die darüber
     *                      informiert, ob noch gewartet wird,
     *                      oder ob es weitergeht.
     */
    @MessageMapping("/{gameId}/setReady")
    @SendTo("/{gameId}/readyState")
    public String processReadyState(@DestinationVariable Long gameId,
                                    Principal principal) {…}

    /***
     * WebSocket - Schnittstelle zum Versenden von Einladungen
     * an andere Spieler. Nach Akzeptieren der Einladung wird eine
     * Nachricht an /{gameId}/gameChanged gesendet.
     *
     * @param gameId    Die Id des Spiels um die Partie zu identifizieren
     * @param principal Der Aufrufer zum Prüfen der Berechtigung
     * @param player    Der Spieler der eingeladen werden soll
     * @return          Das neue Spiel Objekt wo der neue Spieler
     *                  als Teilnehmer dabei ist
     */
    @MessageMapping("/{gameId}/sendInvite")
    @SendTo("/{gameId}/gameChanged")
    public Game sendInvite(@DestinationVariable Long gameId,
                           Principal principal,
                           Player player) {…}

}
```

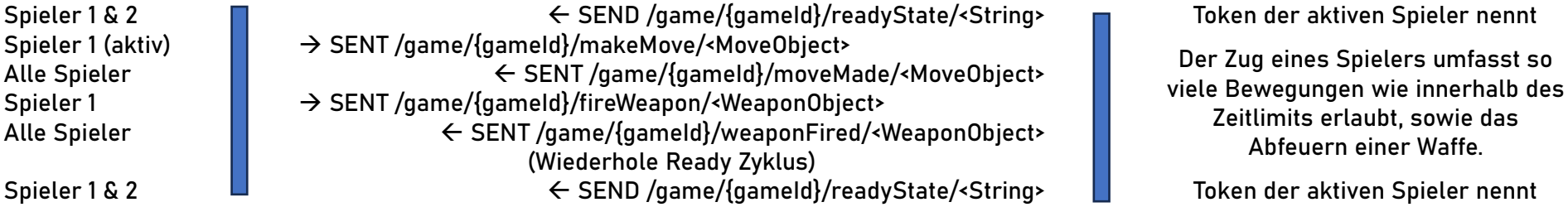
Kommunikation via STOMP / WebSocket

Lobby



GameController

	Client		Server	Bemerkungen
Spieler 1		→POST <url>/game/startNewGame		Eine neue Partie wird erstellt und ein Spielobjekt mit entsprechender Id zurückgegeben
			←<GameObject>	
Spieler 1		→SUBSCRIBE /game/{gameId}/gameChanged		
			←ok /error	Die zur Konfiguration notwendigen Sockets werden abonniert
Spieler 1		→SUBSCRIBE /game/{gameId}/readyState		
			←ok / error	
Spieler 1		→SEND /game/{gameId}/sendInvite/<SpielerObject>		
Spieler 2			← /chat/messages/<MessageObject>	Wird ein Spieler in eine Partie eingeladen, erhält dieser in der Lobby eine entsprechende Nachricht. Wird die Einladung akzeptiert so wird dieser Spieler in den Konfigurationsbildschirm weitergeleitet und alle anderen Spieler werden über den Beitritt benachrichtigt.
Spieler 2 (joins)		→SUBSCRIBE /game/{gameId}/gameChanged		
			←ok /error	
Spieler 2		→SUBSCRIBE /game/{gameId}/readyState/<String>		
			←ok / error	
Spieler 1		← SEND /game/{gameId}/gameChanged/<GameObject>		
Spieler 2		← SEND /game/{gameId}/gameChanged/<GameObject>		
Spieler 1		→ SEND /game/{gameId}/changeGame/<GameObject>		
Spieler 1		← SEND /game/{gameId}/gameChanged/<GameObject>		
Spieler 2		← SEND /game/{gameId}/gameChanged/<GameObject>		
Spieler 1		→ SEND /game/{gameId}/setReady		
Spieler 1		← SEND /game/{gameId}/readyState/<String>		Ready Zyklus, alle Spieler müssen ein „Ready“ Signal senden. Hat der letzte fehlende Spieler das Signal gesendet fährt der Server fort.
Spieler 2		← SEND /game/{gameId}/readyState/<String>		
Spieler 2		→ SEND /game/{gameId}/setReady		
Spieler 1		← SEND /game/{gameId}/readyState/<String>		
Spieler 2		← SEND /game/{gameId}/readyState/<String>		
Spieler 1		→ POST /game/{gameId}/play		Das Spiel wird auf aktiv gesetzt
Spieler 1 & 2		→ SUBSCRIBE /game/{gameId}/weaponFired		
			← ok / error	Entsprechende Sockets werden von allen Spielern abonniert
Spieler 1 & 2		→ SUBSCRIBE /game/{gameId}/moveMade		
			← ok / error	
		(Wiederhole Ready Zyklus)		


Spieler 1 & 2
Spieler 1 (aktiv)
Alle Spieler
Spieler 1
Alle Spieler
Spieler 1 & 2
← SEND /game/{gameId}/readyState/<String>
→ SENT /game/{gameId}/makeMove/<MoveObject>
← SENT /game/{gameId}/moveMade/<MoveObject>
→ SENT /game/{gameId}/fireWeapon/<WeaponObject>
← SENT /game/{gameId}/weaponFired/<WeaponObject>
(Wiederhole Ready Zyklus)
← SEND /game/{gameId}/readyState/<String>
Token der aktiven Spieler nennt
Der Zug eines Spielers umfasst so viele Bewegungen wie innerhalb des Zeitlimits erlaubt, sowie das Abfeuern einer Waffe.
Token der aktiven Spieler nennt