ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук

Департамент программной инженерии

|  |  |
| --- | --- |
| СОГЛАСОВАНО  Научный руководитель  старший преподаватель  департамента больших данных и информационного поиска  факультета компьютерных наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.М. Гринкруг  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. | УТВЕРЖДЕНО  Академический руководитель  образовательной программы  «Программная инженерия»  профессор департамента программной инженерии, канд. техн. наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.В. Шилов  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. |

|  |  |
| --- | --- |
| Подп. и дата |  |
| Инв. № дубл. |  |
| Взам. инв. № |  |
| Подп. и дата |  |
| Инв. № подл |  |

Сервер для мокирования асинхронных протоколов

Текст программы

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.03.07-01 12 01-ЛУ

|  |  |
| --- | --- |
|  | Исполнители  студент группы БПИ152  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** /П.И. Данилин/  **«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  2019 г. |

УТВЕРЖДЕН

RU.17701729. 03.07-01 12 01-1-ЛУ

Сервер для мокирования асинхронных протоколов

|  |  |
| --- | --- |
| Подп. и дата |  |
| Инв. № дубл. |  |
| Взам. инв. № |  |
| Подп. и дата |  |
| Инв. № подл |  |

Текст программы

RU.17701729.03.07-01 12 01-ЛУ

Листов 26

# Аннотация

В данном программном документе приведен текст программы «Сервер для мокирования асинхронных протоколов». В разделе «Текст программы» находится текст программы, распределенный по файлам.

Оформление программного документа «Текст программы» произведено по требованиям ГОСТ 19.401-78 «Текст программы. Требования к содержанию и оформлению».

**Содержание**

[Аннотация 3](#_Toc9706502)

[1. Текст программы 5](#_Toc9706503)

[Routable.scala 5](#_Toc9706504)

[DelayedSource.scala 7](#_Toc9706505)

[JsonUtils.scala 8](#_Toc9706506)

[MockConfiguration.scala 13](#_Toc9706507)

[AdminMockConfigurationController.scala 14](#_Toc9706508)

[MockEndpointManager.scala 15](#_Toc9706509)

[EndpointManagerImpl.scala 16](#_Toc9706510)

[AsyncMock.scala 17](#_Toc9706511)

[Directives.scala 18](#_Toc9706512)

[FutureUtils.scala 18](#_Toc9706513)

[JsonComparator.scala 19](#_Toc9706514)

[Main.scala 20](#_Toc9706515)

[swagger.yaml 21](#_Toc9706516)

# Текст программы

## Routable.scala

**package** paulymorph.mock.configuration.route  
  
**import** akka.NotUsed  
**import** akka.http.scaladsl.model.HttpMethods  
**import** akka.http.scaladsl.server.Directives.\_  
**import** akka.http.scaladsl.server.{Directive0, Route}  
**import** akka.stream.scaladsl.Source  
**import** paulymorph.mock.configuration.\_  
**import** paulymorph.mock.configuration.stub.\_  
**import** paulymorph.mock.configuration.stub.http.\_  
**import** paulymorph.mock.configuration.stub.websocket.WsReaction  
**import** paulymorph.utils.Directives  
  
**trait** Routable[T] {  
 **def** toRoute(value: T): Route  
}  
  
**object** RoutableSyntax {  
  
 **implicit class** RoutableOps[T](value: T) {  
 **def** toRoute(**implicit** routable: Routable[T]) = routable.toRoute(value)  
 }  
  
}  
  
**object** Routable {  
  
 **import** DirectableSyntax.DirectableOps  
 **import** RoutableSyntax.RoutableOps  
  
 **implicit lazy val** stubConfigRoutable: Routable[StubConfiguration] = configuration => {  
 **import** akka.http.scaladsl.server.Directives.\_  
 configuration.stubs.map(\_.toRoute).fold(reject)(\_ ~ \_)  
 }  
  
 **implicit lazy val** responseRoutable: Routable[Response] = {  
 **case** response: SseEventsResponse =>  
 **import** akka.http.scaladsl.marshalling.sse.EventStreamMarshalling.\_  
  
 **import** scala.concurrent.duration.\_  
  
 complete {  
 DelayedSource.createMessageLike(response.events)  
 .map(\_.toSse)  
 .takeWithin(response.timeout.getOrElse(10.minutes))  
 }  
 **case** response: WebSocketEventsResponse =>  
 **import** scala.concurrent.duration.\_  
 **val** source = DelayedSource.createMessageLike(response.events)  
 .map(\_.toWs)  
 **val** incomingMessagesHandler = WsReaction.toFlow(response.reactions)  
 handleWebSocketMessages(incomingMessagesHandler.merge(source).takeWithin(response.timeout.getOrElse(10.minutes)))  
 }  
  
 **implicit lazy val** responseStubRoutable: Routable[ResponseStub] = (stub: ResponseStub) => {  
 **val** predicateDirective = stub.predicates.foldLeft(pass) { **case** (accDirective, predicate) =>  
 accDirective & predicate.toDirective(Directable.predicateDirectable)  
 }  
 **val** responseRoute = Directives.cyclic(stub.responses.map(\_.toRoute))  
  
 predicateDirective {  
 responseRoute  
 }  
 }  
  
 **implicit lazy val** stubRoutable: Routable[Stub] = {  
 **case** stub: ResponseStub => stub.toRoute  
 }  
  
 **implicit lazy val** mockRoutable: Routable[MockConfiguration] = {  
 **case** mock: StubConfiguration => mock.toRoute  
 }  
}  
  
**trait** Directable[T] {  
 **def** toDirective(value: T): Directive0  
}  
  
**object** Directable {  
  
 **import** DirectableSyntax.\_  
 **import** akka.http.scaladsl.server.Directives.\_  
  
 **implicit lazy val** predicateDirectable: Directable[HttpPredicate] = {  
 **case** And(predicates) => predicates.map(\_.toDirective(predicateDirectable)).fold(pass)(\_ & \_)  
 **case** Or(predicates) => predicates.map(\_.toDirective(predicateDirectable)).fold(pass)(\_ | \_)  
  
 **case** Equals(MethodExpectation(expectedMethod)) =>  
 method(HttpMethods.getForKeyCaseInsensitive(expectedMethod).getOrElse(???))  
 **case** Contains(MethodExpectation(expectedMethod)) =>  
 method(HttpMethods.getForKeyCaseInsensitive(expectedMethod).getOrElse(???))  
 **case** StartsWith(MethodExpectation(expectedMethod)) =>  
 method(HttpMethods.getForKeyCaseInsensitive(expectedMethod).getOrElse(???))  
  
 **case** Equals(PathExpectation(expectedPath)) =>  
 path(expectedPath)  
 **case** Equals(BodyExpectation(expectedBody)) => ???  
 **case** Equals(QueryExpectation(expectedQuery)) =>  
 parameterMap.flatMap { actualParams =>  
 validate(expectedQuery.forall(e => actualParams.toSet.contains(e)), **s"Query $**actualParams **did not equal $**expectedQuery**"**)  
 }  
  
 **case** Contains(PathExpectation(expectedPath)) =>  
 extractUri.flatMap { uri =>  
 **val** fullPath = uri.toRelative.path.dropChars(1).toString  
 validate(fullPath.contains(expectedPath), **s"Path $**fullPath **did not contain $**expectedPath**"**)  
 }  
 **case** Contains(BodyExpectation(expectedBody)) => ???  
 **case** Contains(QueryExpectation(expectedQuery)) =>  
 parameterMap.flatMap { actualParams =>  
 validate(expectedQuery.forall {  
 **case** (key, expectedSubstring) =>  
 actualParams.get(key)  
 .exists(\_.contains(expectedSubstring))  
 }, **s"Query $**actualParams **did not contain $**expectedQuery**"**)  
 }  
  
 **case** StartsWith(BodyExpectation(expectedBody)) => ???  
 **case** StartsWith(PathExpectation(expectedPath)) =>  
 extractUri.flatMap { uri =>  
 **val** fullPath = uri.toRelative.path.dropChars(1).toString  
 validate(fullPath.startsWith(expectedPath), **s"Path $**fullPath **did not match prefix $**expectedPath**"**)  
 }  
 **case** StartsWith(QueryExpectation(expectedQuery)) =>  
 parameterMap.flatMap { actualParams =>  
 validate(expectedQuery.forall {  
 **case** (key, expectedPrefix) =>  
 actualParams.get(key)  
 .exists(\_.startsWith(expectedPrefix))  
 }, **s"Query $**actualParams **did not startWith $**expectedQuery**"**)  
 }  
 }  
  
}  
  
**object** DirectableSyntax {  
  
 **implicit class** DirectableOps[T](value: T) {  
 **def** toDirective(**implicit** directable: Directable[T]) = directable.toDirective(value)  
 }  
  
}  
  
**trait** Sourcable[T, E] {  
 **def** toSource(value: T): Source[E, NotUsed]  
}  
  
**object** SourcableSyntax {  
  
 **implicit class** SourcableOps[T](value: T) {  
 **def** toSource[E](**implicit** sourcable: Sourcable[T, E]) = sourcable.toSource(value)  
 }  
  
}

## DelayedSource.scala

**package** paulymorph.mock.configuration.route  
  
**import** akka.NotUsed  
**import** akka.stream.DelayOverflowStrategy  
**import** akka.stream.scaladsl.Source  
**import** paulymorph.mock.configuration.stub.MessageLike  
  
**import** scala.concurrent.duration.\_  
  
**object** DelayedSource {  
 **def** create[T](events: Seq[(T, FiniteDuration)]): Source[T, NotUsed] = {  
 **case class** Accumulator(delay: FiniteDuration, source: Source[T, NotUsed])  
 events.foldLeft(Accumulator(0.millis, Source.empty[T])) { **case** (accumulator, (element, delay)) =>  
 **val** newDelay = accumulator.delay + delay  
 **val** newElementSource = Source.single(element).delay(newDelay, DelayOverflowStrategy.backpressure)  
 Accumulator(newDelay, accumulator.source.concat(newElementSource))  
 }.source  
 }  
  
 **def** createMessageLike[T <: MessageLike](events: Seq[T]): Source[T, NotUsed] = {  
 create(events.map(e => (e, e.delay.getOrElse(0.millis))))  
 }  
}

## JsonUtils.scala

**package** paulymorph.mock.configuration.json  
  
**import** cats.syntax.functor.toFunctorOps  
**import** io.circe.generic.semiauto.{deriveDecoder, deriveEncoder}  
**import** io.circe.syntax.EncoderOps  
**import** io.circe.{Decoder, DecodingFailure, Encoder, HCursor, Json}  
**import** paulymorph.mock.configuration.stub.http.{CompoundHttpPredicate, HttpPredicate, LeafHttpPredicate, RequestExpectation}  
**import** paulymorph.mock.configuration.stub.websocket.{CompoundWsPredicate, WsContains, WsEquals, WsEventPredicate, WsReaction, WsStartsWith}  
**import** paulymorph.mock.configuration.stub.{MockConfiguration, Response, ResponseStub, SseEventsResponse, SseMessage, Stub, StubConfiguration, WebSocketEventsResponse, WsMessage}  
  
**import** scala.concurrent.duration.FiniteDuration  
**import** scala.util.{Failure, Try}  
  
**object** JsonUtils {  
 **implicit lazy val** mockConfigurationEncoder: Encoder[MockConfiguration] = Encoder.instance {  
 **case** configuration: StubConfiguration => configuration.asJson  
 }  
 **implicit lazy val** mockConfigurationDecoder: Decoder[MockConfiguration] =  
 List[Decoder[MockConfiguration]](  
 Decoder[StubConfiguration].widen  
 ).reduceLeft(\_ or \_)  
  
 **implicit lazy val** stubConfigurationEncoder: Encoder[StubConfiguration] = deriveEncoder  
 **implicit lazy val** stubConfigurationDecoder: Decoder[StubConfiguration] = deriveDecoder  
  
 **implicit lazy val** stubEncoder: Encoder[Stub] = Encoder.instance {  
 **case** responseStub: ResponseStub => responseStub.asJson  
 }  
 **implicit lazy val** stubDecoder: Decoder[Stub] =  
 List[Decoder[Stub]](  
 Decoder[ResponseStub].widen  
 ).reduceLeft(\_ or \_)  
  
 **implicit lazy val** responseStubEncoder: Encoder[ResponseStub] = deriveEncoder  
 **implicit lazy val** responseStubDecoder: Decoder[ResponseStub] = deriveDecoder  
  
 **implicit lazy val** predicateEncoder: Encoder[HttpPredicate] = Encoder.instance {  
 **case** predicate: LeafHttpPredicate => predicate.asJson  
 **case** predicate: CompoundHttpPredicate => predicate.asJson  
 }  
 **implicit lazy val** predicateDecoder: Decoder[HttpPredicate] = {  
 **import** paulymorph.mock.configuration.stub.http.\_  
 c: HCursor => {  
 **val** predicates: List[Decoder.Result[HttpPredicate]] =  
 c.get[Seq[RequestExpectation]](**"equals"**).map(e => And(e.map(Equals))) ::  
 c.get[Seq[RequestExpectation]](**"startsWith"**).map(e => And(e.map(StartsWith))) ::  
 c.get[Seq[RequestExpectation]](**"contains"**).map(e => And(e.map(Contains))) ::  
 c.get[Seq[HttpPredicate]](**"or"**).map(Or) ::  
 c.get[Seq[HttpPredicate]](**"and"**).map(And) ::  
 Nil  
  
 **val** resultPredicate = predicates.flatMap(\_.toOption) **match** {  
 **case** single :: Nil => single  
 **case** predicatesList => And(predicatesList)  
 }  
 Right(resultPredicate)  
 }  
 }  
  
 **implicit lazy val** leafPredicateEncoder: Encoder[LeafHttpPredicate] = { predicate =>  
 **import** paulymorph.mock.configuration.stub.http.\_  
 **def** encodeLeaf(key: String): Json =  
 Encoder.encodeMap[String, Json].apply(Map(key -> predicate.requestExpectation.asJson))  
  
 predicate **match** {  
 **case** \_: Equals => encodeLeaf(**"equals"**)  
 **case** \_: StartsWith => encodeLeaf(**"startsWith"**)  
 **case** \_: Contains => encodeLeaf(**"contains"**)  
 }  
 }  
  
 **implicit lazy val** compoundPredicateEncoder: Encoder[CompoundHttpPredicate] = { predicate =>  
 **import** paulymorph.mock.configuration.stub.http.\_  
 **def** encodeCompound(key: String): Json =  
 Encoder.encodeMap[String, Json].apply(Map(key -> predicate.predicates.asJson))  
  
 predicate **match** {  
 **case** \_: Or => encodeCompound(**"or"**)  
 **case** \_: And => encodeCompound(**"and"**)  
 }  
 }  
  
 **implicit lazy val** requestExpectationsEncoder: Encoder[RequestExpectation] = deriveEncoder  
  
 **implicit lazy val** requestExpectationsDecoder: Decoder[Seq[RequestExpectation]] =  
 (c: HCursor) => {  
 **import** paulymorph.mock.configuration.stub.http.\_  
 **val** expectations = c.get[Json](**"body"**).map(BodyExpectation) ::  
 c.get[String](**"path"**).map(PathExpectation) ::  
 c.get[String](**"method"**).map(MethodExpectation) ::  
 c.get[Map[String, String]](**"query"**).map(QueryExpectation) ::  
 Nil  
  
 Right(expectations.flatMap(\_.toOption))  
 }  
  
 **implicit lazy val** responseEncoder: Encoder[Response] = Encoder.instance {  
 **case** sseResponse: SseEventsResponse => sseResponse.asJson  
 **case** wsResponse: WebSocketEventsResponse => wsResponse.asJson  
 }  
 **implicit lazy val** responseDecoder: Decoder[Response] =  
 List[Decoder[Response]](  
 Decoder[SseEventsResponse].widen,  
 Decoder[WebSocketEventsResponse].widen  
 ).reduceLeft(\_ or \_)  
  
  
 **implicit lazy val** sseEventsResponseEncoder: Encoder[SseEventsResponse] =  
 Encoder.forProduct3(**"events"**, **"timeout"**, **"type"**) { resp =>  
 (resp.events, resp.timeout, **"sse"**)  
 }  
 **implicit lazy val** sseEventsResponseDecoder: Decoder[SseEventsResponse] =  
 (c: HCursor) =>  
 **for** {  
 responseType <- c.downField(**"type"**).as[String]  
 events <- c.downField(**"events"**).as[Option[Seq[SseMessage]]]  
 timeout <- c.downField(**"timeout"**).as[Option[FiniteDuration]]  
 result <- responseType **match** {  
 **case "sse"** => Right(SseEventsResponse(events.getOrElse(Seq.empty), timeout))  
 **case** \_ => Left(DecodingFailure(**s"$**responseType **is not sse!"**, c.history))  
 }  
 } **yield** result  
  
 **implicit lazy val** sseEventEncoder: Encoder[SseMessage] = deriveEncoder  
 **implicit lazy val** sseEventDecoder: Decoder[SseMessage] = deriveDecoder  
  
 **implicit lazy val** wsEventsResponseEncoder: Encoder[WebSocketEventsResponse] =  
 Encoder.forProduct3(**"events"**, **"timeout"**, **"type"**) { resp =>  
 (resp.events, resp.timeout, **"websocket"**)  
 }  
 **implicit lazy val** wsEventsResponseDecoder: Decoder[WebSocketEventsResponse] =  
 (c: HCursor) =>  
 **for** {  
 responseType <- c.downField(**"type"**).as[String]  
 events <- c.downField(**"events"**).as[Option[Seq[WsMessage]]]  
 timeout <- c.downField(**"timeout"**).as[Option[FiniteDuration]]  
 reactions <- c.downField(**"reactions"**).as[Option[Seq[WsReaction]]]  
 result <- responseType **match** {  
 **case "websocket"** => Right(WebSocketEventsResponse(events.getOrElse(Seq.empty), reactions.getOrElse(Seq.empty), timeout))  
 **case** \_ => Left(DecodingFailure(**s"$**responseType **is not websocket!"**, c.history))  
 }  
 } **yield** result  
  
 **implicit lazy val** wsEventEncoder: Encoder[WsMessage] = deriveEncoder  
 **implicit lazy val** wsEventDecoder: Decoder[WsMessage] = deriveDecoder  
  
 **implicit lazy val** finiteDurationEncoder: Encoder[FiniteDuration] =  
 Encoder.encodeString.contramap(time => **s"$**{time.toMillis} **millis"**)  
 **implicit lazy val** finiteDurationDecoder: Decoder[FiniteDuration] =  
 Decoder.decodeString.emapTry(string =>  
 string.split(**" "**, 2) **match** {  
 **case** Array(amountString, unitString) =>  
 Try(FiniteDuration(amountString.toLong, unitString))  
 **case** \_ => Failure(**new** IllegalArgumentException(**s"$**string **is not time!"**))  
 }  
 )  
  
 **implicit lazy val** wsReactionEncoder: Encoder[WsReaction] = deriveEncoder  
 **implicit lazy val** wsReactionDecoder: Decoder[WsReaction] = deriveDecoder  
  
 **implicit lazy val** wsEventPredicateEncoder: Encoder[WsEventPredicate] = Encoder.instance {  
 **case** predicate: WsEquals => predicate.asJson  
 **case** predicate: WsContains => predicate.asJson  
 **case** predicate: WsStartsWith => predicate.asJson  
 **case** predicate: CompoundWsPredicate => predicate.asJson  
 }  
 **implicit lazy val** WsEventPredicateDecoder: Decoder[WsEventPredicate] = {  
 **import** paulymorph.mock.configuration.stub.websocket.\_  
 c: HCursor => {  
 **val** predicates: List[Decoder.Result[WsEventPredicate]] =  
 c.get[Json](**"equals"**).map(WsEquals) ::  
 c.get[Json](**"startsWith"**).map(WsStartsWith) ::  
 c.get[Json](**"contains"**).map(WsContains) ::  
 c.get[Set[WsEventPredicate]](**"or"**).map(Or) ::  
 c.get[Set[WsEventPredicate]](**"and"**).map(And) ::  
 Nil  
  
 **val** resultPredicate = predicates.flatMap(\_.toOption) **match** {  
 **case** single :: Nil => single  
 **case** predicatesList => And(predicatesList.toSet)  
 }  
 Right(resultPredicate)  
 }  
 }  
  
 **implicit lazy val** wsEqualsEncoder: Encoder[WsEquals] = deriveEncoder  
  
 **implicit lazy val** WsContainsEncoder: Encoder[WsContains] = deriveEncoder  
  
 **implicit lazy val** wsStartsWithEncoder: Encoder[WsStartsWith] = deriveEncoder  
  
 **implicit lazy val** compoundWsEncoder: Encoder[CompoundWsPredicate] = { predicate =>  
 **import** paulymorph.mock.configuration.stub.websocket.\_  
 **def** encodeCompound(key: String): Json =  
 Encoder.encodeMap[String, Json].apply(Map(key -> predicate.predicates.asJson))  
  
 predicate **match** {  
 **case** \_: Or => encodeCompound(**"or"**)  
 **case** \_: And => encodeCompound(**"and"**)  
 }  
 }  
}  
----WsReaction.scala  
**package** paulymorph.mock.configuration.stub.websocket  
  
**import** akka.NotUsed  
**import** akka.http.scaladsl.model.ws.Message  
**import** akka.stream.scaladsl.{Flow, Source}  
**import** paulymorph.mock.configuration.route.DelayedSource  
**import** paulymorph.mock.configuration.stub.WsMessage  
  
**case class** WsReaction(predicates: Seq[WsEventPredicate], reaction: Seq[WsMessage])  
  
**object** WsReaction {  
 **def** toFlow(reactions: Seq[WsReaction]): Flow[Message, Message, NotUsed] = {  
 Flow[Message].flatMapConcat { message =>  
 reactions.find(\_.predicates.forall(predicate => predicate(message)))  
 .fold(Source.empty[Message]) {  
 reaction => DelayedSource.createMessageLike(reaction.reaction).map(\_.toWs)  
 }  
 }  
 }  
}----WsEventPredicate.scala  
**package** paulymorph.mock.configuration.stub.websocket  
  
**import** akka.http.javadsl.model.ws.TextMessage  
**import** akka.http.scaladsl.model.ws.Message  
**import** io.circe.Json  
**import** paulymorph.utils.JsonComparator  
  
**sealed trait** WsEventPredicate {  
 **def** apply(actualMessage: Message): Boolean  
}  
  
**sealed private**[websocket] **trait** JsonMessagePredicate **extends** WsEventPredicate {  
 **def** comparator: (String, String) => Boolean  
 **def** message: Json  
 **override def** apply(actualMessage: Message): Boolean = actualMessage **match** {  
 **case** textMessage: TextMessage =>  
 JsonComparator.compare(textMessage.getStrictText, message, comparator)  
 **case** \_ => ???  
 }  
}  
  
**case class** WsEquals(message: Json) **extends** JsonMessagePredicate {  
 **override val** comparator: (String, String) => Boolean = \_ == \_  
}  
  
**case class** WsContains(message: Json) **extends** JsonMessagePredicate {  
 **override val** comparator: (String, String) => Boolean = \_.contains(\_)  
}  
  
**case class** WsStartsWith(message: Json) **extends** JsonMessagePredicate {  
 **override val** comparator: (String, String) => Boolean = \_.startsWith(\_)  
}  
  
**sealed trait** CompoundWsPredicate **extends** WsEventPredicate {  
 **def** predicates: Set[WsEventPredicate]  
}  
  
**case class** Or(predicates: Set[WsEventPredicate]) **extends** CompoundWsPredicate {  
 **override def** apply(actualMessage: Message): Boolean =  
 predicates.foldLeft(**true**) { **case** (acc, predicate) =>  
 acc || predicate(actualMessage)  
 }  
}  
**case class** And(predicates: Set[WsEventPredicate]) **extends** CompoundWsPredicate {  
 **override def** apply(actualMessage: Message): Boolean =  
 predicates.foldLeft(**true**) { **case** (acc, predicate) =>  
 acc && predicate(actualMessage)  
 }  
}

## MockConfiguration.scala

**package** paulymorph.mock.configuration.stub  
  
**sealed trait** MockConfiguration {  
 **def** port: Int  
}  
  
**case class** StubConfiguration(port: Int, stubs: Seq[Stub]) **extends** MockConfiguration----Stub.scala  
**package** paulymorph.mock.configuration.stub  
  
**import** akka.http.scaladsl.model.sse.ServerSentEvent  
**import** akka.http.scaladsl.model.ws.TextMessage  
**import** io.circe.Json  
**import** paulymorph.mock.configuration.stub.http.HttpPredicate  
**import** paulymorph.mock.configuration.stub.websocket.WsReaction  
  
**import** scala.concurrent.duration.FiniteDuration  
  
**sealed trait** Stub  
  
**case class** ResponseStub(predicates: Seq[HttpPredicate], responses: Seq[Response]) **extends** Stub  
  
**sealed trait** Response  
  
**trait** MessageLike {  
 **def** data: Json  
 **def** delay: Option[FiniteDuration]  
}  
  
**case class** SseMessage(data: Json,  
 delay: Option[FiniteDuration],  
 eventType: Option[String] = None,  
 id: Option[String] = None,  
 retry: Option[Int] = None) **extends** MessageLike {  
 **def** toSse: ServerSentEvent =  
 ServerSentEvent(data.toString, eventType = eventType, id = id, retry = retry)  
}  
**case class** WsMessage(data: Json, delay: Option[FiniteDuration]) **extends** MessageLike {  
 **def** toWs: akka.http.scaladsl.model.ws.Message = TextMessage(data.toString)  
}  
  
**sealed trait** EventsSequenceLikeResponse **extends** Response {  
 **def** events: Seq[MessageLike]  
 **def** timeout: Option[FiniteDuration]  
}  
  
**case class** SseEventsResponse(events: Seq[SseMessage], timeout: Option[FiniteDuration]) **extends** EventsSequenceLikeResponse  
  
**case class** WebSocketEventsResponse(events: Seq[WsMessage], reactions: Seq[WsReaction], timeout: Option[FiniteDuration]) **extends** EventsSequenceLikeResponse----HttpPredicate.scala  
**package** paulymorph.mock.configuration.stub.http  
  
**import** io.circe.Json  
  
**sealed trait** HttpPredicate  
  
**sealed trait** RequestExpectation  
  
**case class** BodyExpectation(body: Json) **extends** RequestExpectation  
**case class** PathExpectation(path: String) **extends** RequestExpectation  
**case class** MethodExpectation(method: String) **extends** RequestExpectation  
**case class** QueryExpectation(query: Map[String, String]) **extends** RequestExpectation  
  
**sealed trait** LeafHttpPredicate **extends** HttpPredicate {  
 **def** requestExpectation: RequestExpectation  
}  
  
**case class** Equals(requestExpectation: RequestExpectation) **extends** LeafHttpPredicate  
**case class** StartsWith(requestExpectation: RequestExpectation) **extends** LeafHttpPredicate  
**case class** Contains(requestExpectation: RequestExpectation) **extends** LeafHttpPredicate  
  
**sealed trait** CompoundHttpPredicate **extends** HttpPredicate {  
 **def** predicates: Seq[HttpPredicate]  
}  
  
**case class** Or(predicates: Seq[HttpPredicate]) **extends** CompoundHttpPredicate  
**case class** And(predicates: Seq[HttpPredicate]) **extends** CompoundHttpPredicate

## AdminMockConfigurationController.scala

**package** paulymorph.mock.controller  
  
**import** java.util.concurrent.atomic.AtomicReference  
  
**import** akka.Done  
**import** akka.actor.ActorSystem  
**import** akka.http.scaladsl.Http  
**import** akka.http.scaladsl.Http.ServerBinding  
**import** akka.http.scaladsl.model.StatusCodes  
**import** akka.http.scaladsl.server.Route  
**import** akka.stream.Materializer  
**import** com.typesafe.scalalogging.Logger  
**import** paulymorph.mock.configuration.stub.MockConfiguration  
**import** paulymorph.mock.manager.MockEndpointManager  
**import** paulymorph.utils.Directives  
  
**import** scala.concurrent.{ExecutionContext, Future}  
  
**case class** AdminMockConfigurationController(adminPort: Int, endpointManager: MockEndpointManager)  
 (**implicit** actorSystem: ActorSystem,  
 materializer: Materializer) {  
  
 **import** akka.http.scaladsl.server.Directives.\_  
 **import** de.heikoseeberger.akkahttpcirce.ErrorAccumulatingCirceSupport.\_  
 **import** paulymorph.mock.configuration.json.JsonUtils.\_  
  
 **private implicit val** executionContext: ExecutionContext = actorSystem.dispatcher  
  
 **val** adminRoute: Route = pathPrefix(**"mock"**) {  
 (post & entity(as[MockConfiguration])) { stub =>  
 onSuccess(endpointManager.addMock(stub)) {  
 complete(StatusCodes.Created, **s"Created a mock on port $**{stub.port}**"**)  
 }  
 } ~  
 pathPrefix(IntNumber) { port =>  
 get {  
 complete(endpointManager.getMock(port))  
 } ~  
 delete {  
 complete(endpointManager.deleteMock(port))  
 }  
 }  
 }  
  
 **val** swaggerRoute: Route = path(**"swagger"**) {  
 getFromResource(**"swagger/index.html"**)  
 } ~ getFromResourceDirectory(**"swagger"**)  
  
 **private val** atomicBinding = **new** AtomicReference[Option[ServerBinding]](None)  
  
 **def** start: Future[Unit] =  
 Http().bindAndHandle(handler = logDirective(adminRoute ~ swaggerRoute), port = adminPort, interface = **"0.0.0.0"**)  
 .map(binding => atomicBinding.set(Some(binding)))  
 .map(\_ => ())  
  
 **private val** logger = Logger[AdminMockConfigurationController]  
  
 **private val** logDirective = Directives.logRequestResponse(logger)  
  
  
 **def** stop: Future[Unit] = **for** {  
 \_ <- endpointManager.deleteMocks  
 \_ <- atomicBinding.get().fold(Future.successful[Done](Done)) { binding =>  
 binding.unbind()  
 }  
 } **yield** ()  
}

## MockEndpointManager.scala

**package** paulymorph.mock.manager  
  
**import** paulymorph.mock.configuration.stub.MockConfiguration  
  
**import** scala.concurrent.Future  
  
**trait** MockEndpointManager {  
 **def** addMock(mock: MockConfiguration): Future[Unit]  
 **def** deleteMock(port: Int): Future[MockConfiguration]  
 **def** getMock(port: Int): Future[MockConfiguration]  
 **def** deleteMocks: Future[Seq[MockConfiguration]]  
}  
  
**case class** PortAlreadyInUse(port: Int) **extends** RuntimeException {  
 **override def** getMessage: String = **s"Port $**port **is already in use"**}  
  
**case class** NoEndpointStartedOnPort(port: Int) **extends** RuntimeException {  
 **override def** getMessage: String = **s"There is no endpoint on port $**port**"**}

## EndpointManagerImpl.scala

**package** paulymorph.mock.manager  
  
**import** akka.actor.ActorSystem  
**import** akka.http.scaladsl.Http  
**import** akka.http.scaladsl.Http.ServerBinding  
**import** akka.stream.Materializer  
**import** com.typesafe.scalalogging.Logger  
**import** paulymorph.mock.configuration.route.Routable  
**import** paulymorph.mock.configuration.stub.MockConfiguration  
**import** paulymorph.utils.Directives  
  
**import** scala.collection.concurrent.TrieMap  
**import** scala.concurrent.{ExecutionContextExecutor, Future}  
**import** scala.util.Success  
  
**class** EndpointManagerImpl(**implicit** actorSystem: ActorSystem, materializer: Materializer) **extends** MockEndpointManager {  
 **implicit val** ec: ExecutionContextExecutor = actorSystem.dispatcher  
  
 **import** paulymorph.mock.configuration.route.RoutableSyntax.RoutableOps  
  
 **private val** portBindings = TrieMap.empty[Int, (ServerBinding, MockConfiguration)]  
  
 **override def** addMock(mock: MockConfiguration): Future[Unit] = {  
 **if** (portBindings.contains(mock.port))  
 Future.failed(PortAlreadyInUse(mock.port))  
 **else** {  
 **val** route = mock.toRoute(Routable.mockRoutable)  
 **for** {  
 binding <- Http().bindAndHandle(handler = loggerDirective(route), port = mock.port, interface = **"0.0.0.0"**)  
 \_ = portBindings += mock.port -> (binding, mock)  
 \_ = logger.info(**s"Successfully bound on port $**{mock.port}**"**)  
 } **yield** ()  
 }  
 }  
  
 **override def** deleteMock(port: Int): Future[MockConfiguration] = {  
 **for** {  
 bindingOpt <- Future.successful(portBindings.remove(port))  
 (binding, mock) = bindingOpt.getOrElse(**throw** NoEndpointStartedOnPort(port))  
 \_ <- binding.unbind()  
 \_ = logger.info(**s"Endpoint on $**port **successfully stopped"**)  
 } **yield** mock  
 }  
  
 **private lazy val** logger = Logger[EndpointManagerImpl]  
  
 **private lazy val** loggerDirective = Directives.logRequestResponse(logger)  
  
 **override def** getMock(port: Int): Future[MockConfiguration] = **for** {  
 bindingOpt <- Future.successful(portBindings.get(port))  
 (\_, mock) = bindingOpt.getOrElse(**throw** NoEndpointStartedOnPort(port))  
 } **yield** mock  
  
 **override def** deleteMocks: Future[Seq[MockConfiguration]] = {  
 **val** bindings = portBindings.values.toSeq  
 **val** unbindingsFuture = bindings.map { **case** (binding, configuration) =>  
 binding.unbind()  
 .map(\_ => configuration)  
 }  
  
 Future.sequence(unbindingsFuture)  
 .andThen { **case** Success(\_) => portBindings.clear() }  
 }  
}

## AsyncMock.scala

**package** paulymorph  
  
**import** akka.actor.ActorSystem  
**import** akka.stream.ActorMaterializer  
**import** com.typesafe.scalalogging.Logger  
**import** paulymorph.mock.configuration.stub.MockConfiguration  
**import** paulymorph.mock.controller.AdminMockConfigurationController  
**import** paulymorph.mock.manager.EndpointManagerImpl  
  
**import** scala.concurrent.Future  
  
**class** AsyncMock(adminPort: Int, startingMocks: Seq[MockConfiguration] = Seq.empty) {  
 **implicit val** actorSystem = ActorSystem(**"default"**)  
 **implicit val** materializer = ActorMaterializer()  
 **implicit val** executionContext = actorSystem.dispatcher  
  
 **def** start: Future[Unit] = {  
 **for** {  
 \_ <- adminMockManager.start  
 \_ = logger.info(**s"Server successfully started! Visit http://localhost:$**adminPort**/swagger"**)  
 } **yield** ()  
 }  
  
 **def** stop: Future[Unit] =  
 **for** {  
 \_ <- adminMockManager.stop  
 \_ <- actorSystem.terminate()  
 \_ = logger.info(**"AsyncMock successfully stopped"**)  
 } **yield** ()  
  
 **private val** endpointManager = **new** EndpointManagerImpl  
  
 **private val** adminMockManager = AdminMockConfigurationController(adminPort, endpointManager)  
  
 **private val** logger = Logger[AsyncMock]  
}

## Directives.scala

**package** paulymorph.utils  
  
**import** akka.http.scaladsl.model.HttpRequest  
**import** akka.http.scaladsl.server.RouteResult.{Complete, Rejected}  
**import** akka.http.scaladsl.server.directives.LoggingMagnet  
**import** akka.http.scaladsl.server.{Directive0, Route, RouteResult}  
**import** com.typesafe.scalalogging.Logger  
  
**object** Directives **extends** akka.http.scaladsl.server.Directives {  
 **def** cyclic(innerRoutes: Seq[Route]): Route = {  
 **val** iter: Iterator[Route] = Iterator.continually(innerRoutes).flatten  
  
 **val** cyclicRoute: Route = request => {  
 **val** nextRoute = iter.next()  
 nextRoute(request)  
 }  
  
 cyclicRoute  
 }  
  
 **def** logRequestResponse(logger: Logger): Directive0 = {  
 logRequestResult({  
 **def** logRequestAndResponse(req: HttpRequest)(res: RouteResult): Unit = {  
 logger.info(**s"$**{req.method.value} **$**{req.uri}**"**)  
 logger.debug(req.toString)  
 res **match** {  
 **case** Complete(response) => logger.info(**s"$**{response.status} **$**{response.entity}**"**)  
 **case** Rejected(rejections) => logger.info(rejections.toString)  
 }  
 logger.debug(res.toString)  
 }  
  
 LoggingMagnet(\_ => logRequestAndResponse)  
 })  
 }  
}

## FutureUtils.scala

**package** paulymorph.utils  
  
**import** scala.concurrent.{ExecutionContext, Future}  
  
**trait** FutureUtils {  
 **def** optTraverse[T, U](option: Option[T])(f: T => Future[U])  
 (**implicit** ec: ExecutionContext): Future[Option[U]] = {  
 option **match** {  
 **case** Some(value) => f(value)  
 .map(Some.apply)  
 **case** None => Future.successful(None)  
 }  
 }  
  
 **def** eitherTraverse[L, R, U](either: Either[L, R])(f: R => Future[U])  
 (**implicit** ec: ExecutionContext): Future[Either[L, U]] = {  
 either **match** {  
 **case** Left(value) => Future.successful(Left(value))  
 **case** Right(value) => f(value)  
 .map(result => Right(result))  
 }  
 }  
}  
  
**object** FutureUtils **extends** FutureUtils {  
}

## JsonComparator.scala

**package** paulymorph.utils  
  
**import** io.circe.Json  
**import** io.circe.parser.decode  
  
**object** JsonComparator {  
 **def** compare(actual: String, expected: Json, comparator: (String, String) => Boolean): Boolean = {  
 expected.asString **match** {  
 **case** Some(expectedString) => comparator(actual, expectedString)  
 **case** \_ => (  
 **for** {  
 actualJson <- decode[Json](actual)  
 compared = compare(actualJson, expected, comparator)  
 } **yield** compared  
 ).getOrElse(**false**)  
 }  
 }  
  
 **def** compare(actual: Json, expected: Json, comparator: (String, String) => Boolean): Boolean = {  
 (actual, expected) **match** {  
 **case** \_ **if** actual.isString && expected.isString =>  
 comparator(actual.asString.get, expected.asString.get)  
 **case** \_ **if** actual.isObject && expected.isObject =>  
 expected.asObject.get.toVector.foldLeft(**true**) {  
 **case** (**false**, \_) => **false  
 case** (\_, (key, expectedValue)) =>  
 (**for** {  
 actualValue <- actual.hcursor.get[Json](key)  
 compared = compare(actualValue, expectedValue, comparator)  
 } **yield** compared).getOrElse(**false**)  
 }  
 **case** \_ **if** actual.isArray && expected.isArray => ???  
 **case** \_ => actual == expected  
 }  
 }  
}

## Main.scala

**import** com.typesafe.scalalogging.Logger  
**import** paulymorph.AsyncMock  
**import** paulymorph.mock.configuration.stub.MockConfiguration  
  
**import** scala.concurrent.ExecutionContext  
**import** scala.io.Source  
**import** scala.util.Try  
  
**object** Main {  
 **implicit val** ec = ExecutionContext.global  
 **import** paulymorph.mock.configuration.json.JsonUtils.\_  
  
 **def** main(args: Array[String]): Unit = {  
 logger.info(**s"Starting AsyncMock with args $**args**..."**)  
  
 **val** startingMocks = handleCommandLineArguments(args)  
 **val** server = **new** AsyncMock(2525, startingMocks)  
  
 **val** serverLifetime = **for** {  
 \_ <- server.start  
 \_ = logger.info(**"Press ENTER to close application"**)  
 \_ = scala.io.StdIn.readLine()  
 \_ = logger.info(**"Stopping AsyncMock"**)  
 \_ <- server.stop  
 } **yield** ()  
  
 serverLifetime.failed.foreach(exception => {  
 *logger*.error(**s"Fatal error in execution. Exiting..."**, exception)  
 System.exit(1)  
 }  
 )  
 }  
  
 **def** handleCommandLineArguments(args: Seq[String]): Seq[MockConfiguration] = {  
 args **match** {  
 **case** Seq() => Seq.empty  
  
 **case** Seq(fileFlag, filePath, tail@\_\*) **if** List(**"-f"**, **"--file"**).contains(fileFlag) =>  
 **import** io.circe.parser.decode  
 **val** parsedOpt = Try {  
 **val** fileContents = Source.fromFile(filePath).mkString  
 decode[MockConfiguration](fileContents)  
 .getOrElse(**throw new** IllegalArgumentException(**s"$**filePath **is incorrect mock"**))  
 }.toOption.toSeq  
  
 parsedOpt ++ handleCommandLineArguments(tail)  
  
 **case** Seq(unknown, tail@\_\*) =>  
 logger.warn(**s"$**unknown **is not recognized"**)  
 handleCommandLineArguments(tail)  
 }  
 }  
  
 **private val** logger = Logger[Main.**type**]  
}

## swagger.yaml

**openapi**: 3.0.0  
**info**:  
 **title**: AsyncMock management API  
 **description**: API for mocks management.  
 **version**: 0.1.0  
  
**paths**:  
 **/mock**:  
 **post**:  
 **summary**: Creates a mock.  
 **requestBody**:  
 **required**: true  
 **content**:  
 **application/json**:  
 **schema**:  
 **$ref**: **'#/components/schemas/MockConfiguration'  
 responses**:  
 **201**:  
 **description**: The mock was created successfully.  
 **/mock/{port}**:  
 **parameters**:  
 - **in**: path  
 **name**: port  
 **schema**:  
 **type**: integer  
 **example**: 5000  
 **required**: true  
 **description**: Port of the mock  
 **get**:  
 **summary**: Gets current mock at the port.  
 **responses**:  
 **200**:  
 **description**: Successfully got the mock on the specified port.  
 **content**:  
 **application/json**:  
 **schema**:  
 **$ref**: **'#/components/schemas/MockConfiguration'  
 delete**:  
 **summary**: Remove a mock at the port.  
 **responses**:  
 **200**:  
 **description**: Successfully removed the mock on the specified port.  
 **content**:  
 **application/json**:  
 **schema**:  
 **$ref**: **'#/components/schemas/MockConfiguration'  
components**:  
 **schemas**:  
 **MockConfiguration**:  
 **description**: Mock configuration  
 **type**: object  
 **properties**:  
 **port**:  
 **type**: number  
 **example**: 5000  
 **stubs**:  
 **type**: array  
 **items**:  
 **$ref**: **'#/components/schemas/Stub'  
 Stub**:  
 **type**: object  
 **properties**:  
 **predicates**:  
 **description**: Predicates to match the incoming HTTP requests to the server  
 **type**: array  
 **items**:  
 **$ref**: **'#/components/schemas/HttpPredicate'  
 responses**:  
 **type**: array  
 **items**:  
 **$ref**: **'#/components/schemas/Response'  
 HttpPredicate**:  
 **type**: object  
 **anyOf**:  
 - **$ref**: **'#/components/schemas/HttpEquals'** - **$ref**: **'#/components/schemas/HttpContains'** - **$ref**: **'#/components/schemas/HttpStartsWith'** - **$ref**: **'#/components/schemas/HttpAnd'** - **$ref**: **'#/components/schemas/HttpOr'  
 example**:  
 **equals**:  
 **method**: get  
 **startsWith**:  
 **path**:  
 **"foo"  
 HttpEquals**:  
 **type**: object  
 **required**: [equals]  
 **properties**:  
 **equals**:  
 **$ref**: **'#/components/schemas/RequestExpectation'  
 HttpContains**:  
 **type**: object  
 **required**: [contains]  
 **properties**:  
 **contains**:  
 **$ref**: **'#/components/schemas/RequestExpectation'  
 HttpStartsWith**:  
 **type**: object  
 **required**: [startsWith]  
 **properties**:  
 **startsWith**:  
 **$ref**: **'#/components/schemas/RequestExpectation'  
 HttpAnd**:  
 **type**:  
 object  
 **required**: [and]  
 **properties**:  
 **and**:  
 **type**:  
 array  
 **items**:  
 **$ref**: **'#/components/schemas/HttpPredicate'  
 HttpOr**:  
 **type**:  
 object  
 **required**: [or]  
 **properties**:  
 **or**:  
 **type**:  
 array  
 **items**:  
 **$ref**: **'#/components/schemas/HttpPredicate'  
 RequestExpectation**:  
 **type**: object  
 **anyOf**:  
 - **$ref**: **'#/components/schemas/BodyExpectation'** - **$ref**: **'#/components/schemas/PathExpectation'** - **$ref**: **'#/components/schemas/MethodExpectation'** - **$ref**: **'#/components/schemas/QueryExpectation'  
 example**:  
 **body**:  
 **key**: value  
 **path**: path/path  
 **method**: get  
 **query**:  
 **a**: b  
 **BodyExpectation**:  
 **type**: object  
 **required**: [body]  
 **properties**:  
 **body**:  
 **$ref**: **'#/components/schemas/JsonLike'  
 PathExpectation**:  
 **type**: object  
 **required**: [path]  
 **properties**:  
 **body**:  
 **type**: string  
 **MethodExpectation**:  
 **type**: object  
 **required**: [method]  
 **properties**:  
 **method**:  
 **type**: string  
 **enum**:  
 - get  
 - post  
 - put  
 - delete  
 - head  
 - patch  
 - trace  
 **example**:  
 **method**: get  
 **QueryExpectation**:  
 **type**: object  
 **Response**:  
 **$ref**: **'#/components/schemas/EventsResponse'  
 EventsResponse**:  
 **oneOf**:  
 - **$ref**: **'#/components/schemas/SseEventsResponse'** - **$ref**: **'#/components/schemas/WsEventsResponse'  
 properties**:  
 **type**:  
 **type**: string  
 **enum**: [sse, websocket]  
 **discriminator**:  
 **propertyName**: type  
 **mapping**:  
 **sse**: **'#/components/schemas/SseEventsResponse'  
 websocket**: **'#/components/schemas/WsEventsResponse'  
 SseEventsResponse**:  
 **type**: object  
 **properties**:  
 **type**:  
 **type**: string  
 **enum**: [sse]  
 **events**:  
 **type**: array  
 **items**:  
 **$ref**: **'#/components/schemas/SseEvent'  
 timeout**:  
 **$ref**: **'#/components/schemas/Duration'  
 JsonLike**:  
 **oneOf**:  
 - **type**: object  
 - **type**: string  
 - **type**: number  
 **example**:  
 **key**: value  
 **EventLike**:  
 **type**: object  
 **properties**:  
 **data**:  
 **$ref**: **'#/components/schemas/JsonLike'  
 delay**:  
 **$ref**: **'#/components/schemas/Duration'  
 required**: [data]  
 **Duration**:  
 **type**: string  
 **pattern**: **'\d+ (millis?|seconds?|minutes?)'  
 example**: **'2 seconds'  
 SseEvent**:  
 **type**: object  
 **allOf**:  
 - **$ref**: **'#/components/schemas/EventLike'  
 required**: [data]  
 **properties**:  
 **id**:  
 **type**: string  
 **eventType**:  
 **type**: string  
 **WsEventsResponse**:  
 **type**: object  
 **properties**:  
 **type**:  
 **type**: string  
 **enum**: [websocket]  
 **events**:  
 **type**: array  
 **items**:  
 **$ref**: **'#/components/schemas/WsEvent'  
 reactions**:  
 **type**:  
 array  
 **items**:  
 **$ref**: **'#/components/schemas/WsReaction'  
 timeout**:  
 **$ref**: **'#/components/schemas/Duration'  
 WsEvent**:  
 **type**: object  
 **allOf**:  
 - **$ref**: **'#/components/schemas/EventLike'  
 required**: [data]  
 **WsReaction**:  
 **type**:  
 object  
 **properties**:  
 **predicates**:  
 **type**: array  
 **items**:  
 **$ref**: **'#/components/schemas/WsEventPredicate'  
 reaction**:  
 **type**: array  
 **items**:  
 **$ref**: **'#/components/schemas/WsEvent'  
 WsEventPredicate**:  
 **type**: object  
 **anyOf**:  
 - **$ref**: **'#/components/schemas/WsEquals'** - **$ref**: **'#/components/schemas/WsContains'** - **$ref**: **'#/components/schemas/WsStartsWith'** - **$ref**: **'#/components/schemas/WsAnd'** - **$ref**: **'#/components/schemas/WsOr'  
 WsEquals**:  
 **type**: object  
 **required**: [equals]  
 **properties**:  
 **equals**:  
 **$ref**: **'#/components/schemas/JsonLike'  
 WsContains**:  
 **type**: object  
 **required**: [contains]  
 **properties**:  
 **contains**:  
 **$ref**: **'#/components/schemas/JsonLike'  
 WsStartsWith**:  
 **type**: object  
 **required**: [startsWith]  
 **properties**:  
 **startsWith**:  
 **$ref**: **'#/components/schemas/JsonLike'  
 WsAnd**:  
 **type**: object  
 **required**: [and]  
 **properties**:  
 **and**:  
 **type**: array  
 **items**:  
 **$ref**: **'#/components/schemas/WsEventPredicate'  
 WsOr**:  
 **type**: object  
 **required**: [or]  
 **properties**:  
 **or**:  
 **type**: array  
 **items**:  
 **$ref**: **'#/components/schemas/WsEventPredicate'**