Request Interceptor

(ehemals "Request Handler")

Bestimmte, von Clients (z. B. Browser) angefragte Pfade, können durch Plugins überwacht und optional direkt beantwortet werden, ohne die Standardabläufe des Frameworks (wie Projekt und Menüpunkt finden) zu involvieren.

```
fun registerRequestInterceptor(
  pathPattern: String,
  interceptor: RequestInterceptor,
  priority: Int = 50
)
fun registerRequestInterceptor(
  condition: RequestCondition,
  interceptor: RequestInterceptor
)
```

Im einfachsten Fall wird ein regulärer Ausdruck (`pathPattern`) definiert und alle Requests mit passendem Pfad (also dem Text hinter der Domain ab `/`) werden an den `interceptor` gegeben. Mit einer `RequestCondition` kann zusätzlich noch die Domain (auch mittels regulärem Ausdruck) abgefragt werden. Außerdem kann damit nur auf Forwards reagiert (`onlyForwards()`) oder es können Forwards ausgeschlossen werden (`noForwards()`).

Der `interceptor` muss das Interface `com.batix.plugins.RequestInterceptor` implementieren. Hier gibt es drei Methoden, von denen eine oder mehrere überschrieben werden können.

Die `priority` bestimmt die Ausführungsreihenfolge der Interceptors und kann gesetzt werden, wenn man mit einem Interceptor andere beeinflussen will.

`RequestInterceptor`-Interface

> **Thread-Safety**
> Dieselbe Instanz wird für alle Requests verwendet, muss also thread-safe implementiert werden.

Es können drei Methoden implementiert werden:

```
fun handlePre(event: RequestEvent)
fun handlePost(event: RequestEvent)

fun handleFrontendException(event: RequestEvent, exception: FrontendException)
```

Das `RequestEvent` beinhaltet dabei den `event.request: HttpServletRequest` und den `event.response: HttpServletResponse` sowie Methoden, um die folgende Requestbehandlung zu beeinflussen.

`handlePre` wird noch vor allen Framework-Funktionen aufgerufen und bestimmt durch das übergebene `RequestEvent`, ob der Request komplett vom Plugin behandelt wurde und deshalb von folgenden Plugins oder vom Framework nicht weiter beachtet werden soll. Wird auf dem Event `preventDefault` aufgerufen, wird das Framework seine Standardabläufe (wie z. B. Projekt und Menüpunkt finden) nicht durchführen. Ohne den Aufruf läuft der Request normal weiter - in diesem Fall sind Modifikationen an `response` mit Vorsicht zu genießen, da die weiteren Abläufe ggf. zu einem Redirect führen oder gesetzte Werte wieder überschrieben werden können. Wird auf dem Event `stopPropagation` aufgerufen, werden (nach priority) nachfolgende Plugins den Request nicht erhalten und damit auch nicht auf diesen reagieren können.

Der Aufruf von `handlePost` erfolgt, nachdem alle anderen Abläufe im Framework erledigt sind, auch falls es Fehler gab. In dieser Phase könnten schon Teile der Response-Header oder sogar des Response-Bodys geschrieben sein, Modifikationen dort sollten also mit Bedacht durchgeführt werden, wenn überhaupt. Methodenaufrufe auf dem `RequestEvent` zur Beeinflussung des nachfolgenden Ablaufs sind hier wirkungslos - das Framework hat den Request bereits behandelt, und nachfolgende Plugins werden auch nur dann übersprungen, falls in `handlePre` der `stopPropagation`-Aufruf erfolgte.

Mit `handleFrontendException` kann auf Fehler aufgrund verschiedener Dinge, wie z. B. Projekt / Menüpunkt / Datei nicht gefunden, fehlende Authentifizierung oder auch Serverfehler, reagiert werden - wobei der genaue Fehler(-Grund) in `exception` festgehalten ist. Hier kann lediglich mit `preventDefault` die nachfolgende Fehlerbehandlung des Frameworks unterbunden werden - nicht aber die nachfolgender Plugins.

Möchte man die Methoden je Request korrelieren, so bieten sich Request-Attribute an, in denen man seinen State festhalten kann. In der `RequestInterceptor`-implementierenden Klasse selbst können keine Felder o. ä. benutzt werden, da die Instanz der Klasse ja threadübergreifend verwendet wird.

Es werden übrigens nicht nur Frontendaufrufe gematcht. Auch Aufrufe des Backends können überwacht werden. So könnte man dieses z. B. durch eine Prüfung der Client-IP, oder anderen Merkmalen, noch weiter abschotten.

Eine Sonderform von Request Interceptors ist für statischen Content verfügbar.

Priority

Die `priority` bestimmt die Ausführungsreihenfolge der Interceptors, auch über mehrere Plugins hinweg: Je niedriger die Priorität, desto früher wird ein Interceptor aufgerufen. Im Regelfall spielt sie kaum eine Rolle, ist aber entscheidend um zu bestimmen, welche Interceptors denn von Aufrufen von `stopPropagation` betroffen werden sollen.

`stopPropagation` kann aber in jedem Fall nur Interceptors mit einem größeren `priority`-Value betreffen - mehrere Interceptors derselben Priorität können sich nicht gegenseitig abbrechen. Zudem wird sich der Prioritätswert gemerkt, falls in `handlePre` `stopPropagation` aufgerufen wird, und gewährleistet, dass auch das `handlePost` der verhinderten Interceptors nicht aufgerufen wird.

`registerRequestInterceptor( condition: RequestCondition, interceptor: RequestInterceptor)` nimmt keine gesonderte `priority`, da diese intern Teil der `RequestCondition` ist und direkt auf ihr definiert werden kann.

Beispiel

Hier werden drei Request Interceptors registriert. Einer misst die Dauer von Requests, ein Weiterer überprüft Authentifizierung bei API-Zugriffen, ein Letzter führt eine Fehlerstatistik.

```
override fun load() {
  registerRequestInterceptor(
    "^/api/.*",
    AuthInterceptor(),
    RequestCondition.AUTH_PRIORITY
  )

  registerRequestInterceptor("^/api/.*", TimingInterceptor())
  registerRequestInterceptor("^(?!/verwaltung/).*", ErrorMetricsInterceptor())
}
```

Der `AuthInterceptor` prüft bei jedem Request, ob es sich um einen authentifizierten `BxUser` handelt, der einer bestimmten Gruppe angehört und blockt unauthentifizierte Anfragen ab, indem er die Ausführung nachfolgender Interceptors (in diesem Fall `TimingInterceptor` - aber auch andere Interceptors anderer Plugins, die auf dem selben Pfad lauschen) und Framework-Funktionen verhindert. Die `AUTH_PRIORITY` entspricht einem Wert von *15* und liegt damit vor allen anderen registrierten Interceptors. Hier wird zwar nur `handlePre` implementiert, trotzdem wird implizit auch `handlePost` vom `TimingInterceptor` verhindert.

```
import com.batix.Log
import com.batix.ConnectionPool
import com.batix.modul.BxUser
import com.batix.plugins.RequestEvent
import com.batix.plugins.RequestInterceptor
import jakarta.servlet.http.HttpServletResponse

class AuthInterceptor : RequestInterceptor {
  override fun handlePre(event: RequestEvent) {
    val user = BxUser.findInstance(event.request) // User-Instanz
    if (user != null) { // angemeldet
      ConnectionPool.withSystemConnectionDo { conn ->
        if ("17AD26C9C4C" in user.getGroups(conn).map { it.id }) {
          Log.debug("user is authenticated")
          return
        }
      }
    }

    event.preventDefault()
    event.stopPropagation()
    event.response?.sendError(HttpServletResponse.SC_UNAUTHORIZED)
  }
}
```

Der `TimingInterceptor` merkt sich zu Beginn aller Requests, die mit `/api/` anfangen, deren Start in einem Request-Attribut. Wenn der Request fertig ist, wird der Startzeitpunkt wieder ausgelesen, die Dauer berechnet und geloggt. Ohne explizite Angabe einer `priority` erhält er die `RequestCondition.DEFAULT_PRIORITY` von *50*.

```
import com.batix.Log
import com.batix.plugins.RequestEvent
import com.batix.plugins.RequestInterceptor
import java.time.Duration
import java.time.Instant

class TimingInterceptor : RequestInterceptor {
  override fun handlePre(event: RequestEvent) {
    event.request.setAttribute(REQUEST_STARTED_ATTRIBUTE, Instant.now())
    // event.preventDefault() // CMS-Weiterbehandlung stoppen
```

```
    // event.stopPropagation() // Plugin-Weiterbehandlung stoppen
  }

  override fun handlePost(event: RequestEvent) {
    val instant = event.request.getAttribute(REQUEST_STARTED_ATTRIBUTE) as Instant
    val duration = Duration.between(instant, Instant.now())
    Log.debug("request of ${event.request.requestURL} took $duration")
  }

  companion object {
    private const val REQUEST_STARTED_ATTRIBUTE = "timing-interceptor-started"
  }
}
```

`ErrorMetricsInterceptor` überwacht für alle Requests, die **nicht** mit `/verwaltung/` anfangen, aufgetretene Fehler und aktualisiert eine fiktive Statistik.

```
import com.batix.plugins.RequestInterceptor
import com.batix.tags.FrontendException
import com.batix.tags.FrontendExceptionInterface
import com.batix.plugins.RequestEvent

class ErrorMetricsInterceptor : RequestInterceptor {
  override fun handleFrontendException(event: RequestEvent, exception: FrontendException) {
    val errorCode = exception.errorCode
    stats.errors.withInternalCode(errorCode).increment()

    if (errorCode == FrontendExceptionInterface.Code.UNKNOWN_WEB.bxCode) {
      stats.unknownProjects.countDomain(event.request.serverName)
    }
  }
}
```

> **javax / jakarta**
> Ab Framework v3.0 müssen die `jakarta` anstatt der `javax` Klassen verwendet werden.

Statische Ressourcen schützen

Ein anderes Anwendungsgebiet für Request Interceptors ist das Schützen der statischen Ressourcen, die in der Verwaltung unter *Ressourcen > Vorlagen > statische Ressourcen* gepflegt werden.

Hierfür wird einfach ein Request Interceptor für die zu schützenden Pfade / Dateien definiert und die gewünschten User-Checks durchgeführt.

Simples Beispiel:

```
registerRequestInterceptor("^/static/my-project/assets-for-vips/.*$", object :
RequestInterceptor {
  override fun handlePre(event: RequestEvent) {
    val user = BxUser.findInstance(event.request)
    val vipsGroup = UserGroup.findGroup("196243C94D0")
    val allowed = user != null && ConnectionPool.withSystemConnectionDo { conn ->
      user.isInGroup(conn, vipsGroup)
    }

    if (!allowed) {
      event.response.status = HttpServletResponse.SC_UNAUTHORIZED // 401
      event.preventDefault()
    }
  }
})
```

> **Verfügbarkeit**
> Nur verfügbar für Systeme, die in Docker laufen oder speziell konfiguriert sind (`/static/` via Tomcat). Das ist standardmäßig ab Framework Version 2.9 der Fall.