SAIL Events Field

A field for lazy loading data in SAIL interfaces. As SAIL interfaces only display to the user once all variables are populated the initial load time is only as fast as the slowest part. This component allows those slow parts (typically database queries and integration calls) to be loaded later.

This allows the interface to load faster and let the user start working with the information that is available. Once the slow data becomes available the interface will immediately update.

Usage Requirements:

This field necessitates starting a process to execute the slow operation in the background. As such this component is best used in contexts where a process can be started seamlessly as part of the interface loading.

- **Actions** provide an excellent user experience as the original user click can both start an asynchronous process and display an interface.
- **Record dashboards** might provide an acceptable user experience depending on the use case. As the dashboard can't start a process when it loads, you need a way to start one after it's loaded (e.g. as part of expanding a section or clicking a button)
- **Tasks** may provide an excellent user experience. The caveat is if the task can contain stale data (i.e. user is looking at it hours after initial data load). This may require a means to refresh asynchronously by starting a new process.

Use cases:

- To add consistency to the user experience where unreliable data sources are involved. Such as slow APIs or database queries.
- A call center where a specific integration is known to block the interface loading, but the rest of the interface can load now while it waits.
- A user submitting a request that won't return immediately, but will return reasonably swiftly (within 60 seconds). Such as requesting a document to be generated.
- Refresh a stale cache in the background.

Not suitable for:

- Notifying a user of data changed by another user.
- Long-running interfaces where it's not known when the event will arrive (e.g. waiting indefinitely for an incoming phone call or message)
- Persistently streaming event updates to an interface.
- Real-time communication such as chat between users.
- Where significant compromise to application design is required to facilitate starting asynchronous processes (such as not using core Records features).

sailEventsField Function

fn!sailEventsField(topic, numberOfContextKeys, timeout, onContextKeyReceived, onError)

See the process designer description for an explanation of parameters.

Quick Start

1. Deploy `sail-events-servlet.jar`
2. Deploy `sail-events-field.zip`
3. Create table in `jdbc/Appian`

```
CREATE TABLE IF NOT EXISTS `EVT_CONTEXT_KEY` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `timestamp` timestamp NOT NULL DEFAULT current_timestamp(),
  `username` varchar(20) NOT NULL,
  `topic` varchar(36) NOT NULL,
  `context_key` varchar(30) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `timestamp` (`timestamp`)
)
```

12. Create an interface:

```
a!localVariables(
  local!latestContextKey,
  {
    sailEventsField(
      topic: "myrandomtopicuuid",
      timeout: 20,
      onContextKeyReceived: local!latestContextKey
    ),
    a!textField(
      label: "Result",
      value: local!latestContextKey,
      disabled: true,
25.       placeholder: "waiting for context key..."
26.     )
27.   }
   )
```

28. Send an event:

```
29. INSERT INTO `EVT_CONTEXT_KEY` (`username`, `topic`, `context_key`)
    VALUES ('logged.in.user', 'myrandomtopicuuid', 'EVENT_KEY_1');
```

30. Event was not delivered? Make sure you're using the right username and topic. Remember, timing matters - events expire (see FAQ).

How It Works

Using the onContextKeyReceived and onError parameters

The `save!value` of the `onContextKeyReceived` event will contain the context key that was written to the `EVT_CONTEXT_KEY` table.
The `save!value` of the `onError` event may be one of the following:

- `HTTP 408` Request Timeout. Occurs when the `numberOfContextKeys` expected were not received by the time limit.
- `HTTP 429` Too Many Requests. Occurs when the component detects the same user making multiple parallel requests (for example by opening multiple tabs).
- `HTTP 500` Internal Server Error. Returns for unexpected errors. Check `tomcat-stdOut.log` for more details.
- `HTTP 503` Service Unavailable. Occurs when the maximum number of concurrent users (50) has been reached. This is applied per application server (i.e. 150 for 3-node high-availability).

Best Practices

- Generate a unique topic for each request (i.e. use a UUID as the topic)
- Design interfaces expecting that not all event contexts will be delivered when the `sailEventField` loads. There are many reasons this may occur:
    - Network or browser issue.
    - Timeout was reached.
    - Failure writing to `EVT_CONTEXT_KEY` table.

    - The user manually refreshed their interface after receiving some, but not all events.
    - The user is viewing an interface that was originally created minutes, hours or days earlier.
    - The user opened multiple browser tabs that each use `sailEventField`
- Use refresh variables as a backup for when events take longer than the timeout time (see examples).
- Avoid displaying new content above existing content. Loading content above what the user is currently viewing will push it down and cause frustration.
    - Try to load new content from top to bottom so that it's always appended to the page.
    - Considering showing a placeholder that's the same size as the content when it's available.
- You have three events available. If you have more than three things to notify the interface about then group them together into a single event.
- Never use context keys as query identifiers. Using them as such will expose a potential security risk.

Constraints

- The `EVT_CONTEXT_KEY` table MUST be accessible through the JNDI name `jdbc/Appian` (which is the default business schema name on Appian Cloud)
- The `jdbc/Appian` data source MUST be defined in Tomcat, NOT the Admin Console.
- Topics are user specific. Two users with identical topics are treated separately. It is not possible to have multiple users listening to the same topic.
- A maximum of three events can be consumed by a `sailEventField`. Each event triggers a server-side call to `onContextKeyReceived`. This is designed to reduce the number round-trips to the server-side.
- The component will stop listening for new events once all events have been received or the timeout has been reached.
- Multiple `sailEventFields` cannot be loaded concurrently by the same user. Only one request at a time is permitted.

Frequently asked questions

Q: Does this work in high-availability environments?

A: Yes. Each application server will read from `EVT_CONTEXT_KEY`, which allows users to be connected to any application server to receive event notifications.

Q: Is database polling for events scalable?

A: Yes. The `EVT_CONTEXT_KEY` table is polled once per application server regardless of the number of users waiting for events. If there are no users waiting for events then the database is not polled. Polling occurs every 2.5 seconds while there is one or more users waiting.

Q: What happens to undelivered events?

A: There may be many reasons an event is not delivered (see best practices section for possible reasons). Topics have a two minute time-to-live (TTL) starting from the time the first event is sent to the topic. After this time, the topic expires and undelivered events are discarded.

Q: Does the `sailEventField` have to be loaded by the user before events are written to the database?

A: No. Events can be written to `EVT_CONTEXT_KEY` before the user loads the interface. However, you must consider the events time-to-live (see question above). Wait too long and the events on the topic will expire.

Q: Can the progress bar color be changed?

A: Yes. The bar uses the accent color defined by your environment and site branding.

Examples

Using a refresh variable when an event is not delivered before the timeout

```
a!localVariables(
  local!contextKeys,
  local!eventFieldError,

  /* copy this refreshVariable for each event and change KEY_1 */
  local!mydata: a!refreshVariable(
    value: rule!myFastQueryToGetMyData(...),
    refreshInterval: if(
      and(
        /* don't refresh if we already have data */
        a!isNullOrEmpty(fv!value),
        /* do refresh if a timeout occured or we have the context key */
        or(
          not(a!isNullOrEmpty(local!eventFieldError)),
          contains(local!contextKeys, "KEY_1")
        )
      ),
      0.5,
      null
    )
  ),

  {
    sailEventsField(
      topic: "myrandomtopicuuid",
      timeout: 10,
      numberOfContextKeys: 1,
      onContextKeyReceived: a!save(
        local!contextKeys,
        append(local!contextKeys, tostring(save!value))
      ),
      onError: local!eventFieldError
    ),
    a!textField(
      label: "Result",
      value: local!mydata,
      disabled: true,
      placeholder: "waiting for data ..."
    )
  }
)
```

```
INSERT INTO `EVT_CONTEXT_KEY` (`username`, `topic`, `context_key`)
VALUES ('logged.in.user', 'myrandomtopicuuid', 'EVENT_KEY_1');
```