Eclipse Scout Job API

since Eclipse Scout Neon

Agenda

- Functionality
- Terms related to Job API
- New concepts (RunMonitor, RunContext)
- Job factories
- Scheduling a job
- Await a job’s completion
- Listen for job lifecycle events

Functionality

- based on Java Executors framework;
- job manager is application scoped;
- Provides support …
  - for one-shot or periodic actions;
  - for delayed execution;
  - for mutual exclusion among jobs;
  - to listen for job lifecycle events based on filters;
  - to wait for jobs to complete based on filters;
  - to visit running jobs based on filters;

Terms related to Job API

JobManager
Central point to schedule jobs

Job
Work in the form of a Runnable or Callable

Future
Result of an asynchronous computation

JobInput
Job parametrization

RunContext
Semantic execution context

RunMonitor
Cancellation

RunContext

[new concept]

[not directly related to Job API]

- ➔ used to run code on behalf of some semantic context, e.g. to run code as a specific user, with a different Locale, in a separate transaction, …;
- ➔ code is run in the current thread, meaning that the caller is blocked until completion;
- ➔ facilitates propagation of state among different threads;
- ➔ is associated with a *RunMonitor to* query for cancellation;

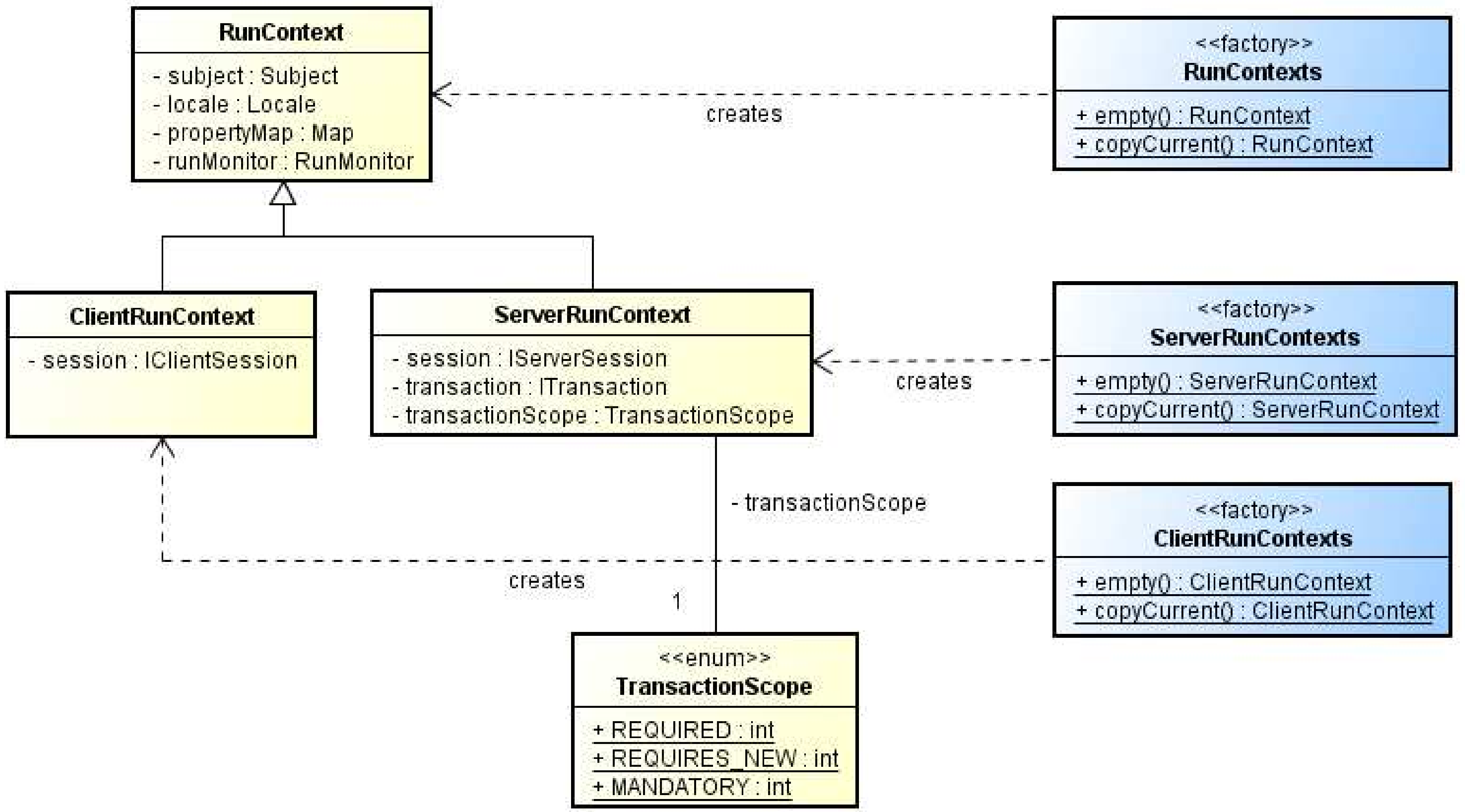
Before Scout Neon, tight coupling of context and job (runNow)

RunContext

```
RunContexts.copyCurrent()
  .withSubject(john)
  .withLocale(Locale.US)
  .run(new IRunnable() {

     @Override
     public void run() throws Exception {
       // executed as ‘john’ with Locale US
     }
});
```

Different RunContexts



Go transactional before Neon

```
Subject john = ...;
Locale oldLocale = LocaleThreadLocal.get();

LocaleThreadLocal.set(Locale.US);
try {
  new ServerJob("...", ServerSession.get(), john) {

    @Override
    protected IStatus runTransaction(IProgressMonitor monitor) {
      // executed as ‘john’ with Locale.US and a new TX
      return Status.OK_STATUS;
    }
  }.runNow(new NullProgressMonitor());
}
finally {
  LocaleThreadLocal.set(oldLocale);
}
```

Go transactional since Neon

```
ServerRunContexts.copyCurrent()
  .withSubject(john)
  .withLocale(Locale.US)
  .withTransactionScope(TransactionScope.REQUIRED)
  .run(new IRunnable() {
     @Override
     public void run() throws Exception {
       // executed as ‘john’ with Locale US and the same TX
     }
});
```

- TransactionScope.REQUIRES_NEW (by default)
- TransactionScope.REQUIRED
- TransactionScope.MANDATORY

JTA naming

Accessing data of RunContext

– **session:**	ISession.CURRENT.get()
– **transaction:**	ITransaction.CURRENT.get()
– **subject:**	Subject.getSubject(AccessController.getContext())
– **locale:**	NlsLocale.CURRENT.get()
– **propertyMap:**	PropertyMap.CURRENT.get()
– **runMonitor:**	RunMonitor.CURRENT.get()
– **servletRequest:**	HttpServletRoundtrip.CURRENT_HTTP_SERVLET_REQUEST.get()
– **servletResponse:**	HttpServletRoundtrip.CURRENT_HTTP_SERVLET_RESPONSE.get()

RunMonitor

[new concept]

[not directly related to Job API]

- ➔ provides consistent cancellation support;
- ➔ code running within RunContext or job can always query its cancellation status via:

```
RunMonitor.CURRENT.get().isCancelled()
```

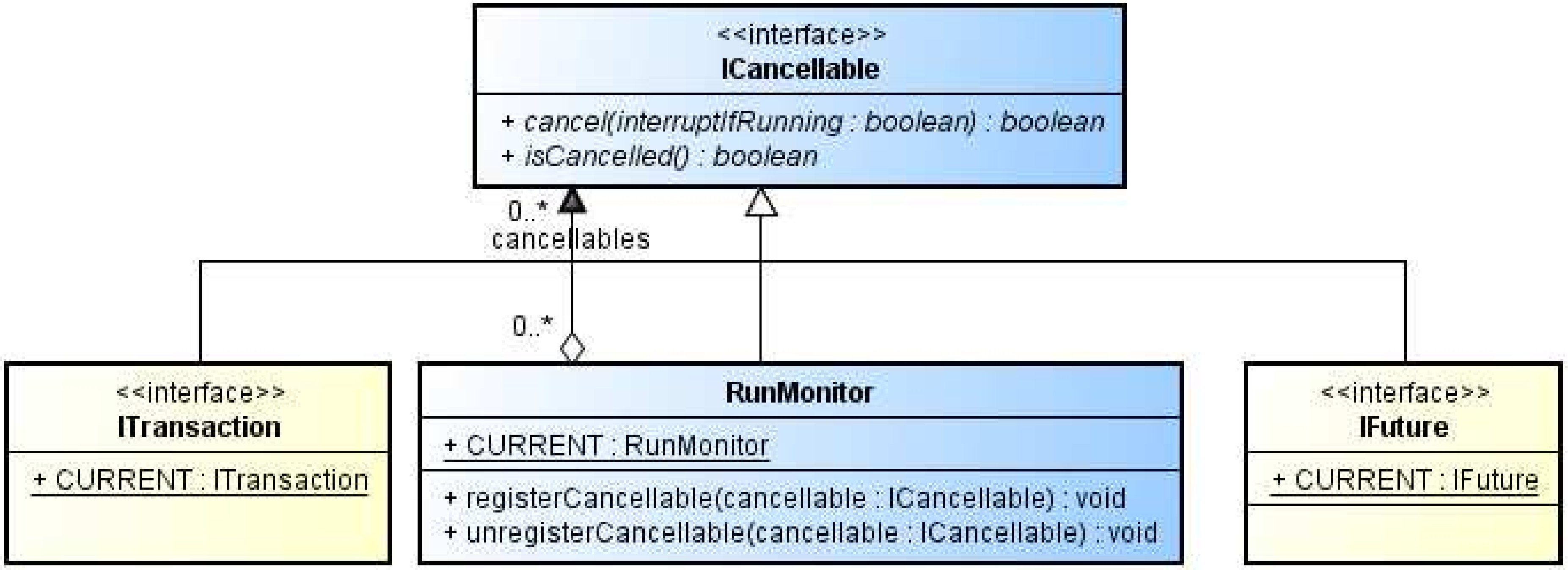
- ➔ allows registration of *Cancellables* like *Transaction*, *Future*, ***RunMonitor***, and others;
- ➔ allows creation of a monitor hierarchy to support nested cancellation;
- ➔ nested cancellation works top-down, and not bottom-up;

JobInput

- ➔ contains meta information about a job;
- ➔ tells the job manager how to run a job

```
Jobs.newInput(RunContexts.copyCurrent())
        .withExpirationTime(2, TimeUnit.SECONDS)
        .withName("data processing")
        .withThreadName("processor")
        .withLogOnError(true)
        .withMutex(Object.class);
```

RunMonitor and ICancellables



<<interface>>
ICancellable
+ cancel(interruptIfRunning : boolean) : boolean
+ isCancelled() : boolean
0..*
cancellables
0..*
<<interface>>
ITransaction
+ CURRENT : ITransaction
RunMonitor
+ CURRENT : RunMonitor
+ registerCancellable(cancellable : ICancellable) : void
+ unregisterCancellable(cancellable : ICancellable) : void
<<interface>>
IFuture
+ CURRENT : IFuture

Job Factories

➔ exists only to conveniently schedule jobs, like proper RunContext or scheduling instructions;

➔ are simply delegates to BEANS.get(IJobManager.class).schedule(...);

Jobs.schedule(...)
optional RunContext

new JobEx(...).schedule()

ServerJobs.schedule(...)
requires ServerRunContext

new ServerJob(...).schedule()

ClientJobs.schedule(...)
requires ClientRunContext

new ClientAsyncJob(...).schedule()

ModelJobs.schedule(...)

- requires ClientRunContext
- to interact with Scout client model
- serial execution

new ClientSyncJob(...).schedule()

Schedule jobs

Run a job

```java
Jobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
      // do something
  }
}, Jobs.newInput(RunContexts.copyCurrent()));
```

```java
Jobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
      // do something
  }
});
```

Schedule jobs

Run a client job

```
ClientJobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
     // do something
  }
}, ClientJobs.newInput(ClientRunContexts.copyCurrent()));
```

```
ClientJobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
     // do something
  }
});
```

Schedule jobs

Run a model job

```
ModelJobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
     // do something
  }
}, ModelJobs.newInput(ModelRunContexts.copyCurrent()));
```

```
ModelJobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
     // do something
  }
});
```

Schedule jobs

Run a server job

```java
ServerJobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
     // do something
  }
}, ServerJobs.newInput(ServerRunContexts.copyCurrent()));
```

```java
ServerJobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
     // do something
  }
});
```

Schedule jobs

Run a job that returns a result

```
IFuture<Void> future = Jobs.schedule(new IRunnable() {

  @Override
    public void run() throws Exception {
      // do something
    }
});
```

Schedule jobs

Run a delayed job

```
Jobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
    // do something in 10 seconds
  }
}, 10, TimeUnit.SECONDS);
```

Schedule jobs

Run a job with another Locale

```
RunContext ctx = RunContexts.copyCurrent().withLocale(Locale.US);

Jobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
    // do something
  }
}, Jobs.newInput(ctx));
```

Schedule periodic jobs

There are two kinds of periodic jobs:

- ➔ at fixed rate
- ➔ with a fixed delay

Run a periodic action at fixed rate

```
Jobs.scheduleAtFixedRate(new IRunnable() {

  @Override
  public void run() throws Exception {
    // is run every 15 seconds
  }
}, 0, 15, TimeUnit.SECONDS, Jobs.newInput(RunContexts.copyCurrent()));
```

Schedule jobs with a mutex

Run jobs in sequence

1

```
Object mutex = new Object();
RunContext ctx = RunContexts.copyCurrent();
```

2

```
Jobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
    // running job 1
  }
}, Jobs.newInput(ctx).mutex(mutex));
```

3

```
Jobs.schedule(new IRunnable() {

  @Override
  public void run() throws Exception {
    // running job 2
  }
}, Jobs.newInput(ctx).mutex(mutex));
```

Await a job’s completion

- ➔ a job can be awaited for on its Future or on the job manager;
- ➔ support for a maximal timeout to wait;
- ➔ a job is ‘done’ once completed or cancelled;

Await a job’s completion

Await job’s completion

```
IFuture<String> future = Jobs.schedule(new Callable<String>() {

  @Override
  public String call() throws Exception {
    return "done";
  }
});

// Blocks current thread until completed or cancelled
future.awaitDone();

// Blocks for a maximal time to get the result
future.awaitDone(1, TimeUnit.MINUTES);

// Blocks and gets the result
String result = future.awaitDoneAndGet();

// Blocks for a maximal time to get the result
String result = future.awaitDoneAndGet(1, TimeUnit.MINUTES);
```

Await a job’s completion

Wait in another thread

```
future.whenDone(new IDoneCallback<String>() {

  @Override
  public void onDone(DoneEvent<String> event) {
    String result =  event.getResult();
  }
});
```

Example: Suspend model thread

```
// running in model thread

final IBlockingCondition bc = Jobs.getJobManager().createBlockingCondition(true);
ClientJobs.schedule(...).whenDone(new IDoneCallback() {

  @Override
  public void onDone(DoneEvent event) {
    bc.setBlocking(false);
  }
});

bc.waitFor(); // allow other model jobs to run
```

Await for multiple jobs to complete

Await for multiple jobs

```
Filter filter = ServerJobs.newFutureFilter()
                          .andMatchCurrentSession()
                          .andMatchNameRegex(Pattern.compile(".*store.*"))
                          .andMatch(new IFilter<IFuture<?>>() {
  @Override
  public boolean accept(IFuture<?> future) {
    return true; // some other criterion
  }
});

Jobs.getJobManager().awaitDone(filter, 1, TimeUnit.MINUTES);
```

Listen for lifecycle job events

Await for multiple jobs

```
Filter filter = ServerJobs.newEventFilter()
                          .andMatchCurrentSession()
                          .andMatchEventTypes(JobEventType.ABOUT_TO_RUN);

Jobs.getJobManager().addListener(filter, new IJobListener() {

  @Override
  public void changed(JobEvent event) {
    // do something
  }
});
```

Thank you

@EclipseScout

daniel.wiehl@bsi-software.com