Information Systems Institute

Distributed Systems Group (DSG)
VL Distributed Systems Technologies SS 2013 (184.260)

Assignment 2

Submission Deadline: 10.5.2013, 18:00

General Remarks

- Group work is not allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the TUWEL forum) are allowed, but the code has to be written alone. If we find that two students submitted the same, or very similar assignments, these students will be graded with 0 points (no questions asked). We will use automated plagiarism checks to compare solutions. Note that we will also include the submissions of previous years in our plagiarism checks.
- No deadline extensions are given. Start early and, after finishing your assignment, upload your submission as a zip file to TUWEL. If you think that you will be hard-pressed making the deadline you should upload a first version well before time runs out! We will grade whatever is there at the deadline, there is no possibility to submit later on.
- Make sure that your solution compiles and runs without errors. If you are unsure, test compiling your submission on different computers (e.g., one of the ZID lab computers). Preferred target platform for the entire lab (all 3 assignments) is JDK 7[1]. If you decide to develop your code using JDK 6, some tweaks (endorsed libraries) may be required to get the project running under this version. Before you submit, make sure to test your solution with JDK 7.
- The assignment project is set up using Apache Maven2[2]. We provide a project template that contains some code interfaces and JUnit tests which will assist you in developing your code. Please stick exactly to the provided interfaces as we will check your solutions in an automated test environment. The Maven project is split up into multiple modules which correspond to the different sub-parts of the assignments. **Note:** If all unit tests are passing in your solution, it does *not* necessarily mean that you will receive all the points. We will perform additional code checks and run tests that are not provided in the template. The tests included in the template should merely help you get started and assist you in developing your solution. If you want to further increase your test coverage, you may also add new unit tests, but this is optional. Do not modify any of the pre-defined tests - in any case, we will check your code with the original tests from the template.
- The root folder of the template contains the Maven metadata file (`pom.xml`) and submodule directories for the three parts of this assignment (`ass2-ejb`, `ass2-ws`, `ass2-di`). Extract these directories to your root directory from assignment 1 and simply overwrite the old `pom.xml` file from assignment 1 with the new `pom.xml` from the template of assignment 2. The dependencies should then be automatically set up by Maven and you can re-use your code (e.g., domain model) from assignment 1.
- Before solving the tasks concerning Enterprise Java Beans, we recommend reading Part IV of the Java EE 6 Tutorial[3]. If you are already familiar with EJB 3.0, this link[4] may give you a good impression of the new features in EJB 3.1 (find parts 1-4 of this series linked in the reference section there). However, note this has been published as a preview: not every feature mentioned made it into the final specification.
- Use MySQL 5.1+[5] as your database management system.
- We expect (as can be seen in the persistence-unit configuration of persistence.xml) that you setup a database `dst` that can be accessed by user `root` (without a password). You may of course change the settings of the configuration for your work at home, but please reset them to the original values in your submitted solution (and make sure it still works). Again: make sure that all settings are as expected before you submit!
- Please make sure to add reasonable logging output to help us keep track of what your solution does. No debug output is very bad, and too much (e.g., many screen pages) is just as bad. Aim for a good middle ground, which allows us to check your solution quickly.

[1] `http://www.oracle.com/technetwork/java/javase/downloads/`
[2] `http://maven.apache.org/`
[3] `http://download.oracle.com/javaee/6/tutorial/doc/bnblr.html`
[4] `http://www.theserverside.com/news/1363649/New-Features-in-EJB-31-Part-5`
[5] `http://dev.mysql.com/downloads/mysql/5.1.htm#downloads`

A. Code Part

Command for building/deploying parts 1.-2.:	`mvn install -Pass2-deploy`
Command for testing part 1. (in separate terminal):	`mvn install -Pass2-ejb`
Command for testing part 2. (in separate terminal):	`mvn install -Pass2-ws`
Command for building/testing part 3.A:	`mvn install -Pass2-di`
Command for building/testing part 3.B:	`mvn install -Pass2-di-agent`

1. Enterprise Java Beans 3.0 (22 Points)

Your task is to develop an enterprise application for the grid management system introduced in assignment 1, using Enterprise Java Beans (EJB). Reuse your entity classes from task 1 (basic mapping and inheritance), and add the required JPA annotations to the `Computer` entity, which you have previously mapped using XML. Code from any other tasks is no longer required. The Maven profile `ass2-deploy` included in the template automatically starts an embedded GlassFish server and deploys the required beans and artifacts. *Note: Due to the high resource consumption of the server, we recommend increasing the JVM's Heap space and the Permanent Generation space using the `MAVEN_OPTS` environment variable (e.g.: export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=1G").*

Make reasonable decisions concerning the type of session bean (stateful, stateless, singleton) to use for each task, and whether the beans should be remotely or locally available. Follow the principles of minimal visibility and minimal accessibility.

1.a. Session beans (13 Points)

- **Create a bean for managing prices (PriceManagementBean)**:

  Our system requires a manageable pricing model for the charges of using the Grid. We assume that the user has to pay two different fees for every assigned job. As a sort of down payment, the user first pays a small fee when assigning a job. The purpose of this fee from the Grid provider's viewpoint is to hedge against the risk of non-payment. Hence, new users have to pay more than long-term users who have already proven to be reliable. In particular, this fee decreases with increasing number of jobs a user has submitted (and paid) in the past. For the second fee, the user's account is debited with the variable price for the job's execution time (after the job has finished).

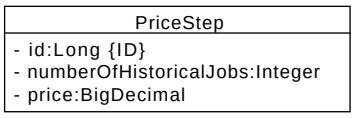

Figure 1: PriceStep Entity

The PriceManagerBean is a helper to adminster the "steps" of the price curve. This bean provides a method to store the price steps in the database, according to the number of jobs the user has previously executed. (e.g.: 30€ for less than 100 executed jobs, 15€ for 100-1000 jobs, 5€ for 1000-5000 jobs ...). The *PriceStep* entity that should be stored is depicted in Figure 1.

In addition to saving price steps, the PriceManagementBean should also offer a method to retrieve price steps, i.e., get the fee for a given number of executed jobs. This bean only serves as a helper for the beans you will implement next. All prices should be retrieved from the database once the server initializes the application and after that stored in memory (to avoid time-consuming database reads at runtime). At this point, creating the required persistent entity should be straightforward. The concrete implementation of the bean should be designed by you. In the end, the bean has to provide a possibility to set and get price steps (i.e., the fee for a given number of historical jobs).

Note that especially the last method (retrieving fees) may get called quite often, so you should keep performance in mind and think about the default behavior of concurrency and transactions managed by the container. However, as a change of prices is not expected to happen all that often, you can directly store new values in the database (do not forget to update your in-memory data structures, which you have loaded at application server startup, though!).

- **Create a bean for general management concerns (GeneralManagementBean)**:

  For now, this bean only has to provide a way to set prices using the bean you just created. We will extend this bean at a later point. However, note that all methods the bean will provide will share no state and will be invoked independently of each other. This bean should also be invokable by the client directly.

- **Create a bean that allows users to assign jobs for several grids (JobManagementBean)**:

  First of all this bean provides a method for the user to login with username and password.

  For more convenience the system provides the possibility to assign several jobs and store them in a single transaction. To do this the user can add jobs for a single grid to a temporary job list by specifying the id of the grid, the number of CPUs, the workflow and parameters (as list) of the job. Think of the temporary job list as a 'shopping cart' for jobs: users add jobs to the list one after another (possibly over a longer timespan) and submit them all in one go.

  When the bean receives requests to add jobs to the temporary job list, it first has to check if there are enough free computing resources (i.e., CPUs) for this grid left. To that end, check if the sum of the CPUs of all free computers in the grid is larger than (or equal to) the CPUs necessary for the job. If this is not the case, the job cannot be scheduled now, and the user is informed (throw a meaningful custom exception). Jobs can be assigned to multiple computers (e.g., a job that requires 6 CPUs could be scheduled to 2 machines with 4 and 2 CPUs, respectively); however, at most one job is executed on each computer at a time. If it is possible to execute the job, the bean has to assign it to concrete computers. In this assignment we will ignore complex scheduling issues. You can simply query all free computers from the database and start assigning free computers at random until the sum of CPUs of all assigned computers is equal to (or larger than) the number of CPUs required by the job.

  After the list has been submitted to the system, all jobs in the list are started immediately (set the executions start field to the current time and the status to `SCHEDULED` – it is not possible to submit jobs where the scheduling starts in the future). When submitting, make sure to check again if the jobs can still be scheduled the way you planned (since the system has many users it is possible that another user was scheduling jobs in parallel using the same computers). Think of a suitable protocol to achieve this. If you find out that it is not possible anymore to execute any of the jobs, notify the user with an exception (and make sure you do not store any computer assignment to the database), as above.

  You should also provide a way to remove all jobs for a specific grid from the temporary job list, as well as a method to get the current list of temporary job assignments.

  Note that no data is written to the database before the temporary job list is finally submitted. So provide a method to do the final submission. Before the final submission, the user has to login at some point in the conversation. When the assignments are finally submitted, store the data (jobs and executions) to the database only if all chosen computers are available. Make sure this method executes transactionally secure, so that no data is written to the database if something goes wrong (i.e.: computers are not available). If the submission was successful, discard the bean. Otherwise, throw meaningful exceptions so that the user can react to this and modify the job assignment (in case of an exception the bean should not be discarded!).

  Note: When running the tests, you may encounter a "javax.ejb.NoSuchObjectLocalException: Invalid Session Key" error message in the server log output. You can safely ignore this exception - it is triggered during a test which ensures that the bean is correctly discarded after submitting the jobs.

1.b. Timer Service and asynchronous method invocations (6 Points)

- **Implement a timer service used for simulation (SimulatorBean)**

  Up to this point we only assigned jobs to the grid. Now, in order to test our solution, we need to simulate that jobs are actually going to finish at some point. We use a timer service to achieve this. The timer service periodically (every 15 seconds) takes the jobs that are running and completes them. A job is considered running if its start time is prior to the current time and its end time is `null`. To complete a running job, set the status (`FINISHED`), end date, and execution time.

- **Provide a method to retrieve the bill of a user**

  Now you should extend the GeneralManagementBean that we have started earlier. Implement a method which can be used to retrieve the total bill of a user (given by username). That means that for each of this user's finished but unpaid jobs compute the costs for the execution (with the grid's `costsPerCPUMinute`) and sum them up. Additionally, add the scheduling costs (the static costs incurred for assigning the jobs in the first place). Use your PriceManagementBean to calculate the scheduling costs, considering the users' discount (if they have a membership for a certain grid). Use the class `BillDTO` to return the bill. The bill contains the total price, the price per job, the setup costs and execution costs, as well as the number of computers that have been used per job. As soon as the bill for one job is finished, you can set the payment status of this job to paid.

  The costs and discount (if applicable) should be calculated as follows: 1) determine the user's discount (d); 2) for each (unpaid) job j in the bill, do: 2.1) query for the total number of already paid jobs (p); 2.2) determine the setup costs c_s from the PriceManagementBean using the parameter p; 2.3) determine the execution costs c_e based on the execution time and the grid's `costsPerCPUMinute`, 2.4) determine the total job costs $c_j = (c_s + c_e) * (1 - d)$; 2.5) set the "paid" status of job j to `true`; 2.6) persist job j to the database. Make sure to round the cost values to two decimal places (e.g., 123.45). **Note:** When looping over the jobs, the value of p should steadily increase (otherwise there could be a price disadvantage if users submit multiple jobs in batches). Think of an appropriate way to achieve this.

  As collecting this data may take some time, the method should be executed asynchronously. The client simply invokes this method, and immediately gets control back. This way, the client can continue processing while the calculation is running, and receives the collected data asynchronously when it's finished. Check out the possibilities that EJB 3.1 provides for this purpose.

1.c. Audit-Interceptor (3 Points)

- **Develop an audit interceptor for the JobManagementBean (AuditInterceptor)**:

  Since the JobManagementBean is essential to our system, we now implement some simple logging to get some insight into the internal workings of the bean. We implement our logger as an audit interceptor. The interceptor should persist the data contained in `IAuditLog` and `IAuditParameter`: invocation time, method name, parameters (index, class and value) and result value (or exception value in case of failure). You may simply invoke the toString() method of objects to convert results to persistable strings (but make sure to check for null values). Note that transactions are usually rolled back in case of an exception, and that this behavior would also influence our interceptor by default. Therefore, check how to bypass this behavior to be able to persist the audit even in case of a failure (e.g., if a job cannot be scheduled). Add a method to the GeneralManagementBean to retrieve all these audits.

1.d. Client application

Some simple client code for testing your EJB application is already included in the template, see folder `ass2-ejb/src/test/java`. You may optionally extend these tests with additional classes. You should provide reasonable output so we can easily keep track of what is going on in your application.

2. Web Services (12 Points)

While EJBs are an important technology of enterprise distributed computing, their usability as integration technology (for instance, between applications of business partners in a value chain, or between applications of different departments) is very limited. For such tasks, SOAP-based Web services have emerged as the current de facto standard. SOAP-based Web services allow heterogeneous applications (i.e., applications written in different programming languages and running on different platforms) to communicate via standard HTTP, across company (and, hence, firewall) borders.

Service Implementation (8 Points)

You should now develop a simple SOAP and WSDL-based service, which (anonymous) external users can invoke to find some base statistics about the utilization of grids. The Web service should provide a operation that returns the finished executions of a grid. The operation takes two parameters: a grid name (String), and the maximum number of executions to return (int). The service returns the executions of this grid using the data transfer object (DTO) named `StatisticsDTO` (DTOs are used to avoid exposing the internal data layer to external parties!). `StatisticsDTO` contains the grid name and a list of executions, each containing the start and end date, plus total number of CPUs used by this execution.

Develop the service using JAX-WS on top of a new EJB (select the correct type of bean for this job), the JobStatisticsBean. Use the JAX-WS reference implementation, which is part of your JDK and available in Glassfish. You can find simple examples on how to create JAX-WS services on the web. For more complex tasks, the JAX-WS specification[6] is the best place to look (scroll to the appendix, *Annotations*).

These are the technical specifications for your service:

- Annotate your service using the constants provided in `dst.ass2.ejb.ws.Constants`.
- The endpoint of your service should be `http://localhost:8080/StatisticsService/service`, consequently, the WSDL contract of your service should be located at `http://localhost:8080/-StatisticsService/service?wsdl`.
- Your service should make use of WS-Addressing for message routing (see here[7] for an example). Assign explicit input, output and fault actions.
- The Web service request and response objects are provided as interfaces (`IGetStatsRequest` and `IGetStatsResponse` in package `dst.ass2.ejb.ws`), and the factory class `WSRequestFactory` is used to instantiate service requests. Provide your implementation of the interfaces in a separate package `dst.ass2.ejb.ws.impl` and return new request instances in the factory method. JAX-WS uses the Java Architecture for XML Binding (JAXB) for (de)serializing the service requests and responses. Since JAXB cannot handle interfaces automatically, you need to provide a JAXB XML Adapter for the request and response objects of the service. Study the JAXB adapter mechanism and provide two implementations of the class `javax.xml.bind.annotation.adapters.XmlAdapter` to correctly marshal/unmarshal request and response objects.
- The "grid name" service parameter should be transported as a header parameter (i.e., the String should be transmitted in the header of the SOAP message, instead of the body). The service response should be transported in the message body, as usual.
- If the passed grid name could not be found in your database, send back a fault message (a SOAP fault) of type UnknownGridFault.

After building and deploying your service, you should be able to access the WSDL contract at the location specified above. You can test your service without writing code using the soapUI tool[8].

Simple tests for the Web service are already included in the template, see folder `ass2-ws/src/test/java`. You may want to take a look at class `dst.ass2.ws.WebServiceUtils`, which conveniently constructs a

[6] http://download.oracle.com/otndocs/jcp/jaxws-2.2-mrel3-evalu-oth-JSpec/
[7] http://jax-ws.java.net/jax-ws-21-ea3/docs/wsaddressing.html
[8] http://www.soapui.org/

Web service proxy for a given Java interface and the WSDL location of the service. Again, feel free to optionally extend the template with your own test classes.

A Peek Under the Cover (4 Points)

So far, we have seen two quite different remoting paradigms in action (EJB remoting and SOAP-based Web services). For this task, you need to take a look under the cover of these technologies. Use a network sniffer, for instance Wireshark[9], to save the client/server interactions for one sample EJB remote invocation and one Web service invocation. Store the sent and received messages to the folder `remoting_artifacts` in your project. Additionally, download the WSDL contract of your Web service and store it to the same directory. Study these artifacts. During the practice lessons, you should be able to discuss what is going on in some detail.

3. Dependency Injection (10 Points)

Dependency Injection is a very important and omnipresent feature to achieve Inversion of Control in modern frameworks and application servers (e.g., Spring[10], EJB[11], CDI[12]). We will now take a detailed look at this technology and implement a simple custom Dependency Injection Controller using annotations. All code for this task has to be put into the `ass2-di` subfolder. The solution for this task has no direct relationship to the grid computing case study.

Task A: Standalone Injection Controller (6 Points)

Your task is to create a thread safe implementation of the supplied `dst.ass2.di.IInjectionController` interface (**take a look at it before you continue reading**):

- All classes of which objects should be initialized or injected using the controller must be annotated by a `Component` annotation. It has to be possible to specify the scope of a component: `SINGLETON` (only one instance is created within the controller and shared between injected objects) or `PROTOTYPE` (a new instance is created every time one is requested). In case the controller is advised to initialize (i.e., the initialize() method is called by the user) an object of a singleton component it already knows an instance of, it should throw an `InjectionException`.
- All Component-annotated classes must have an id, which is annotated as `ComponentId`. The id is unique (in the injector's scope) and of type `Long`. The id field is set by the injector if the object was successfully initialized. If no id is present or the id variable has the wrong type, throw an `InjectionException`. Note that all objects within an inheritance hierarchy must have the same id! (For instance, consider a class `A`, a class `B` that extends `A`, and assume that both `A` and `B` define a `ComponentId`.)
- Every field (inherited, public/private and so on) of a component that is annotated by `Inject` has to be processed. It has to be possible to define whether the injection is `required` (if false, no exception is raised if it is not possible to set this field) and to specify a concrete subtype that should be instantiated (`specificType`). This specific type is optional (if not present, the declared type is used for injection).
- It is not required to deal with circular dependencies. However, you have to completely initialize hierarchically composed objects and report any naming ambiguities that occur. Wrap checked exceptions in InjectionException.

The test classes provided in the template cover a good portion of the required functionality, but we invite you to come up with additional examples or specialized corner cases.

[9] http://www.wireshark.org/
[10] http://www.springsource.org/
[11] http://www.oracle.com/technetwork/java/javaee/ejb/index.html
[12] http://seamframework.org/Weld

Task B: Transparent Injection Controller (4 Points)

So far, to use our dependency injector framework we always have to write the same lines of code to create instances and then let the controller initialize them. To remove this requirement, we are now going to perform code instrumentation on the bytecode level. Use bytecode manipulation to insert a code snippet into each constructor of a `Component` annotated class in which you use an `IInjectionController` instance to initialize the object. Note that it might be necessary to modify the implementation you wrote before - however, it is not necessary that both execution modes work in parallel.

Study the java.lang.instrument[13] package description and implement a `ClassFileTransformer` that modifies the byte code using the Javassist[14] library (the required dependency is already part of the template project). Read the tutorial to understand the concepts of Javassist. If you run the `ass2-di-agent` Maven profile, a jar library will be created and made available for the template project. Study the entire transparent injection mechanism, and take a close look at the `pom.xml` files in the root directory of the template, particularly the `javaagent:...` argLine and the `Premain-Class` manifest entry. During the discussion sessions, you should be able to explain the end-to-end process in detail.

The code snippet in Listing 1 illustrates the controller's functionality and how it is used in both tasks (see next page).

[13] http://java.sun.com/javase/6/docs/api/java/lang/instrument/package-summary.html
[14] http://www.csg.is.titech.ac.jp/ chiba/javassist/

```
@Component(scope = ScopeType.PROTOTYPE)
public class ControllerWithInjections {

     @ComponentId
     private Long id;
     @Inject(specificType = SimpleInterfaceImpl.class)
     private SimpleInterface si;

     public void callSi() {
          si.fooBar();
     }
     public static void taskA() {
          IInjectionController ic = ...;
          ControllerWithInjections cwi =
               new ControllerWithInjections();
          ic.initialize(cwi);
          cwi.callSi(); // output expected
     }
     public static void taskB() {
          ControllerWithInjections cwi =
               new ControllerWithInjections();
          cwi.callSi(); // output expected
     }
}

public interface SimpleInterface {
     void fooBar();
}

@Component(scope = ScopeType.SINGLETON)
public class SimpleInterfaceImpl implements SimpleInterface {

     @ComponentId
     private Long id;

     public void fooBar() {
          System.out.println("[SimpleIntefaceImpl] id: " +
               id + " fooBar called!");
     }
}
```

Listing 1: Sample Components for Dependency Injection

B. Theory Part

The following questions will be discussed during the practice lesson. At the beginning of the each lesson we hand out a list where you can specify which questions you have prepared and are willing to present. We will then select students at random who checked a question to discuss the question (you know the procedure from your math courses). If you are asked to discuss a question but fail to provide a correct and well-founded answer, you will lose **all** points for the theory part of this assignment.

4. EJB Lifecycles (1 Points)

Explain the lifecycle of each bean type defined in the EJB 3.1 specification. What optimizations can the EJB container perform for the respective type? Also think about typical use cases the respective bean type provides to the developer.

5. Dependency Injection (2 Points)

Explain the way dependency injection is performed by the EJB container. What kind of resources may be injected into a bean, and what are the different annotations that can be used?

6. Java Transaction API (2 Points)

The EJB architecture provides a mechanism for distributed transactions. Explain the two ways how transactions can be defined. How is the concept of distributed transactions accomplished behind the scenes, i.e. what tasks have to be performed by the EJB container?

8. Remoting Technologies (1 Points)

Compare EJB remoting and Web services. When would you use one technology, and when the other? Is one of them strictly superior? How do these technologies relate to other remoting technologies that you might know from other lectures (for instance, Java RMI, CORBA, or even socket programming)?