Information Systems Institute

Distributed Systems Group (DSG)
VL Distributed Systems Technologies SS 2011 (184.260)

Assignment 3

Submission Deadline: 31.5.2011, 18:00

General Remarks

- Group work is not allowed in the lab. You have to work alone. Discussions with colleagues (e.g., in the forum[1]) are allowed but the code has to be written alone.
- No deadline extensions are given. Start early and after finishing your assignment upload your submission as a zip file in the DSG-Tool[2]. 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, theres no possibility to submit later on.
- Make sure that your solution includes all the libraries and dependencies and it compiles without errors. If we cannot compile your solution you cannot get all the points.
- Before solving the tasks concerning **Java Server Faces 2.0**, we recommend reading Part II of the Java EE 6 Tutorial[3]. Another very useful tutorial is provided here[4]. Facelets are best described in the developer documentation[5].
- You should use Java 1.6.14+[6], MySQL 5.1[7] as your database management system and GlassFish v3[8] as your application server.
- For your solution, use the provided project stubs:

  All needed libraries are already included if you want to use another one, feel free to integrate it. 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 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). Build scripts for ant[9] are already included, you may alter them, but the submitted solution has to be compilable and runnable with the predefined targets.

[1] https://www.infosys.tuwien.ac.at/teaching/courses/dst/forum/
[2] https://stockholm.vitalab.tuwien.ac.at/dsg-teaching-web/student
[3] http://download.oracle.com/javaee/6/tutorial/doc/bnadp.html
[4] http://www.coreservlets.com/JSF-Tutorial/jsf2/
[5] http://facelets.java.net/nonav/docs/dev/docbook.html
[6] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[7] http://dev.mysql.com/downloads/mysql/5.1.html#downloads
[8] http://glassfish.java.net/downloads/v3-final.html
[9] http://ant.apache.org/

A. Code Part

1. Web Application (11 Points)

In this task we are going to build a web application for the users of the grid example, which has already been used in assignment 1 and 2. Put your code into the **1_jsf** project directory. Please start by studying the build file to understand the deployment process.

In this task you should use Java Server Faces 2.0 with Facelets to create a web interface for the grid management system. Note that in this task you may neither use any special framework like Spring or Seam nor any additional component libraries. The build file expects that you put your facelets into **web/view** and your managed beans and Enterprise Java Beans into **web/src**. You can of course reuse any code you've written so far. All the configuration has already been done for you, and all required libraries are already added. You can type **ant deploy** from scratch to deploy the project and open http://localhost:8080/dst3 to see the start page. It is up to you whether you use managed beans or Enterprise Java Beans to implement the business logic of your web application. You may also decide to use CDI[10] instead of the common JSF 2.0 annotations to manage your beans. **However, your solution should behave exactly as specified**.

In your Web application you have to implement the following use cases. You should develop a central home page (**home.xhtml**) from which all other pages implementing the required functionality are linked. In addition, each of these pages should itself contain a link back to the home page for fast navigation. All messages displayed to the user should be specified in a **message.properties** file (comes bundled with the template). Your interface does not need to look marvelous. Ultimately it's the functionality that counts. However, make sure the interface is intuitively usable.

1. **Add some test data**:

   Provide a button on the initial page (**home.xhtml**) to add at least two grids to the database. One grid offers 2 dual-core and the other 3 quad-core computers.

   After clicking this button, display a message about the succeeded operation. It must not be possible to click this button and add the same data again.

2. **Remove stored data**:

   Provide another button on the same page (**home.xhtml**) to remove all data that may get stored using this web application (i.e., grids, jobs, users and related entities). Again, display a message concerning the operations result when rendering the page. After removing all data, clicking the button from step 1. is possible again, but clicking the remove data button is not. That is, at any time only one of these buttons is active while the other one is disabled.

3. **View all grids**:

   Create a new page (**overview.xhtml**) that displays all grids stored in the database. To this end, create a (view) table that includes the following information: id, name, location, costsPerCPUMinute and the provided amount of free cpus.

4. **Register a new user**:

   On this page (**register.xhtml**), the user can enter all data required to create a new account. The password needs to be entered twice to check for typos. Concerning other input fields, make meaningful assumptions regarding their validation (including address and bank account information). All fields have to be required inputs. Clicking the register-button should result in storing a new user to the database followed by a rendering of **home.xhtml**, informing the user that registration completed successfully. In case anything goes wrong (passwords do not match, username already registered, no input for a required field etc.) provide a meaningful error message and let the user correct the inputs made so far.

[10] http://seamframework.org/Weld

5. **Login as user**:

   The user can login on a separate page (**login.xhtml**) providing his username and password. The user is then forwarded to **home.xhtml**. In any case, provide meaningful messages concerning the operations result. After the login, the links to the register or login page should no longer be visible.

6. **Logout**:

   Clicking this link (which is only visible in case of a successful login) results in logging out the user. After this operation the links to the register and login pages become visible again.

7. **Add jobs to the temporary job list**:

   For each grid in the table created in step 3., provide a button to add a job (Figure 1). Clicking this button results in a forward to a new page (**job.xhtml**).

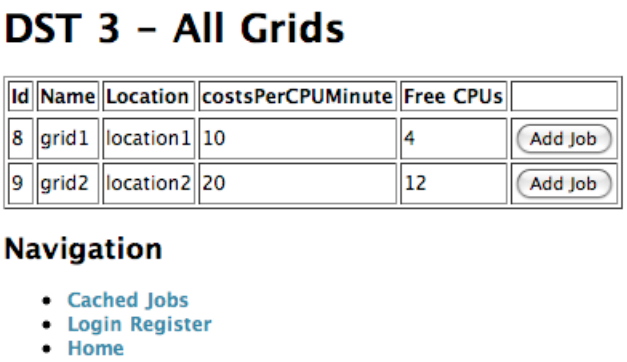


Figure 1: Overview

On this page (**job.xhtml**) the user can enter all data required to add a new job to the temporary job list (Figure 2), excluding the grid id (the grid id depends on the button that the user has clicked). To keep things simple we assume that a job has at most 5 parameters. Additionally, provide a button to add the job to the temporary job list.

When the user adds the jobs, make sure that enough CPUs are actually available and render the **overview.xhtml** page.

The user may add jobs to several grids to the temporary job list and may also add jobs for the same grid again and again - this simply sums up the amount of jobs for this grid in this list. Note that this operation does not depend on the login status of the respective user, i.e., a login is not required to perform this operation and add jobs to the temporary job list. Provide meaningful output messages concerning the operation's result.

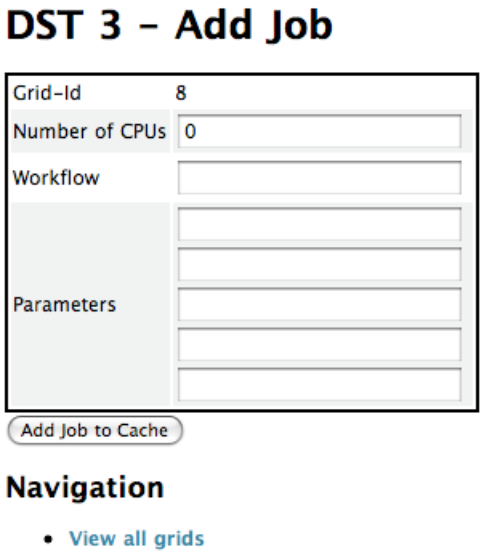


Figure 2: *Add Job* View

8. **The temporary job list**:

   On this page (**assignment.xhtml**), display all grids (by id) and the respective amount of jobs the user has added by the use case described in step 7. Add a button to each table row to remove all jobs for the respective grid from the temporary job list (Figure 3). Last but not least, provide a button to submit all job assignments.

   Do not forget to check again if the jobs can still be scheduled the way you planned (somebody might have submitted an assignment in the meantime!). After executing all required database operations, clear the temporary job list, render **home.xhtml** and print a success message to the user. Using this button requires a logged in user. However, a login must have no influence on the current content of the users cache, i.e.: a user can add all jobs, log in, and then submit the order.

Figure 3: *Temporary Job List* View

9. **Find jobs**:

   As our last task we want to create a page to find all assigned and not processed jobs for a specified grid.

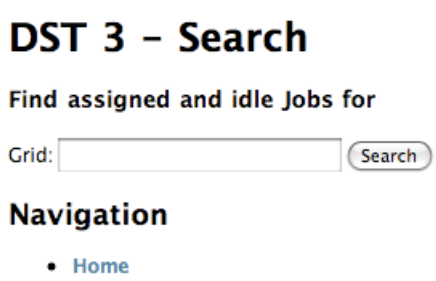
   - To do this we first create a JAX-WS Web Service[11] based on a Stateless Session Bean. This service provides a method to find the jobs. Do not directly return entity bean instances, but

[11] http://download.oracle.com/javaee/6/tutorial/doc/bnbor.html

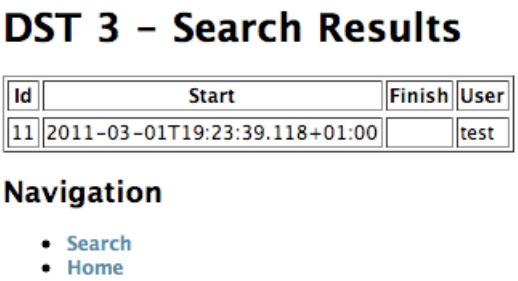
create simple data transfer objects (DTOs) which contain only the following information: id, start-time, finish-time of the job and the username of the user who assigned the job.

You can find the location of your deployed service using Glassfish's administrator console at http://localhost:4848/.

- Now create a bean which acts as the client to access your web service. To generate the required client stubs to access your web service use the wsimport[12] command. It is part of the JDK 6 (e.g. `wsimport -Xnocompile -s src http://localhost:8080/{...}`).
- Provide a page (**search.xhtml**) where the user can enter the name of the grid and a button to start the search (Figure 4a). When the user clicks on the button, check if a grid was entered. If not, display an error message. If the user provided a grid name, make a forward to a new page (**result.xhtml**) to display the search result (Figure 4b). You need to use the Web service here, it is not OK to access the database directly.



(a) search.xhtml



(b) result.xhtml

Figure 4: *Find Jobs* Views

[12] http://download.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html

2. Dependency Injection (10 Points)

Since Dependency Injection (Inversion of Control) is a very important and omnipresent feature of modern frameworks and application servers (e.g., Spring[13], EJB[14], CDI[15]), we will now implement a simple custom Dependency Injection Controller using annotations. All code for this task has to be put into the **2_injection** subfolders. The solution for this and the next task has no relationship to the grid computing case study.

2.a. Standalone injection controller (6 Points)

Your task is to create a thread safe implementation of the supplied `dst3.depinj.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 has to be 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`.
- 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.
- Put your code into the **injector** project.

2.b. Transparent injection controller (4 Points)

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 use some bytecode manipulation (or code instrumentation). 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 - it's not necessary that both execution modes work in parallel.

Study the java.lang.instrument[16] package description and implement a `ClassFileTransformer` that modifies the byte code using the Javassist[17] library (the required .jar file is already part of the template project). Read the tutorial to understand the concepts of Javassist. Put your code into the **agent** project (a jar library will be created and made available for the sample project). Complete the **dist** target to set the premain class attribute.

The following code snippet (Listing 1) illustrates the controller's functionality and how it is used in task 2.a. (`taskA()`) and 2.b. (`taskB()`).

[13] http://www.springsource.org/
[14] http://www.oracle.com/technetwork/java/javaee/ejb/index.html
[15] http://seamframework.org/Weld
[16] http://java.sun.com/javase/6/docs/api/java/lang/instrument/package-summary.html
[17] 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: Code Snippet

You also have to provide a **sample application** that shows the full palette of features of your dependency injection controller. Put your code into the **sample** project. Complete the **run** or **run-with-agent target** (depending on what you have solved).

3. Dynamic Plugin Loading (8 Points)

Another important aspect of modern application servers is the dynamic loading and deployment of plugins or applications. This feature is also a nice demonstration for using reflection and class loading at runtime, so we will again implement a (simplified) custom solution of our own. Put the code for this task into the **3_plugins** subprojects.

- **3.a. Plugin executor (4 Points)**

  Implement the `IPluginExecutor` interface. This is the main component responsible for executing plugins. It has to monitor (i.e., repeatedly list the contents of) several directories to detect whether new **.jar** files were copied to these directories or existing .jar files were modified. The executor then scans the file and looks for classes that implement the `IPluginExecutable` interface. If some plugin executable is found, the executor spawns a new thread and calls its execute method (usage of thread pools is recommended). Take care of class loading: there must not be any problem with the concurrent execution of different plugins containing classes with equal names. Also make sure to free all acquired resources after the execution of a plugin has been completed.

  Put your code into the **loader** project. Complete the **run** target (in your main method you should start scanning for plugins in the **plugins** directory and stop when enter is hit).

- **3.b. Injection class loader (4 Points)**

  Now that our plugin executor works, combine the plugin loading with the **transparent** dependency injection from task 2. This time you should not use the java agent mechanism, but implement a custom class loader that is responsible for loading the classes of a single plugin (a .jar file). When the class has to be loaded, you again have to modify the bytecode using the Javassist[18] library. Think of and implement a way to take care of singletons (regarding memory leaks and overlapping) in this environment!

  Put your code into the **loader** project.

You again have to provide a **sample application** in which you implement at least 2 `IPluginExecutables` that need some time to finish. If you implemented the injection class loader you also have to show this (you may copy the sample from task 2). Put your code into the **sample** project. By using the **dist** target a jar file is automatically built and copied to the **loader/plugins** directory.

[18] http://www.csg.is.titech.ac.jp/ chiba/javassist/

B. Theroy 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. JSF lifecycle and Phase Listeners (2 points)

JSF decomposes a request into multiple phases. Explain this lifecycle and the responsibilities of every single phase. How do you rate the value of phase listeners that may be registered by web applications?

5. CDI (1 Point)

Explain the different built-in scopes supported by CDI (Context and Dependency Injection). What is the difference between a session and a conversation? How can a developer control a conversation using CDI? Also think of situations that would require developing new scopes.

6. Class loading (1 point)

Explain the concept of class loading in Java. What different types of class loaders do exist and how do they relate to each other? How is a class identified in this process? What are the reasons for developers to write their own class loaders?

7. AOP (2 points)

Explain the concept of Aspect Oriented Programming (AOP). Think of typical usage scenarios. Does EJB make use of AOP? How is bytecode manipulation related to this topic?