CS 548—Fall 2017
Enterprise Software Architecture and Design
Assignment Six—Service-Oriented Architecture

In this assignment, you define a SOA for the domain-driven design that you developed in the previous assignment. You will define data transfer objects (DTOs) for your SOA. You will define service façades that provide access to your domain model in terms of DTOs transferred to and from clients. You will define a Web service (SOAP and WSDL) interface for clients that want to access your application over the Web.

The Maven POM file for the ClinicApp module lists the components in the enterprise app that is deployed. The list of enterprise-specific artifacts listed in the POM file now expands to include those artifacts added by this assignment:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-ear-plugin</artifactId>
   <configuration>
      <defaultLibBundleDir>lib</defaultLibBundleDir>
      <modules>
         <webModule>
            <groupId>edu.stevens.cs548</groupId>
            <artifactId>ClinicSoapWebService</artifactId>
            <contextRoot>/clinic-soap</contextRoot>
         </webModule>
         <ejbModule>
            <groupId>edu.stevens.cs548</groupId>
            <artifactId>ClinicService</artifactId>
         </ejbModule>
         <ejbModule>
            <groupId>edu.stevens.cs548</groupId>
            <artifactId>ClinicInit</artifactId>
         </ejbModule>
      </modules>
   </configuration>
</plugin>
```

Data Transfer Objects (DTOs)

You are provided with an Eclipse Maven DTO project. Use this to define the DTOs for your SOA, by defining an XML schema for these DTOs and generating the JAXB classes for the DTOs using a JAXB Maven plugin. There are DTOs for patient and provider objects, as well as for treatment objects. The patient and provider DTOs should include a list of treatment identifiers for the treatments that they are related to. There are three different form of treatment DTOs for treatments, one for each form of treatment (surgery, drug treatment, radiology). You can define treatment DTOs using one element with a choice element, or one element per form of treatment DTO using element substitution. See below how to do this in Java, using JAXB.

In the previous assignment, you used the visitor pattern to encapsulate treatments in the patient and provider objects, to enforce the aggregate pattern: Client logic for visiting a treatment would be defined in terms of a visitor object that clients provide to the API for your DDD. In this assignment, the service façades of the SOA that you define are the clients for the API you defined in the previous assignment. Your service facades define visitor objects that construct treatment DTOs from the underlying PDOs, for example:

```
package edu.stevens.cs548.clinic.service.dto.treatment;

// Generated by JAXB
@XmlRootElement(name = "treatment-dto")
public class TreatmentDto {
    ...
    protected DrugTreatmentType drugTreatment;
    protected RadiologyType radiology;
    protected SurgeryType surgery;
}

public class TreatmentExporter implements ITreatmentExporter<TreatmentDto> {
    private ObjectFactory factory = new ObjectFactory();
    public TreatmentDto exportDrugTreatment
            (long tid, String diagnosis, String drug, float dosage) {
        dto = factory.createTreatmentDto();
        dto.setDrugTreatment(factory.createDrugTreatmentType());
        ...
        return dto;
    }
    public void exportSurgery (…) { ... }
    public void exportRadiology(…) { ... }
}
```

You should define both Java classes and XML schemas for your DTOs. One way to do this is to define a JAXB project in Eclipse that includes the class definitions, with JAXB annotations for example to define the namespaces, and then generate the XML schemas from the JAXB-annotated Java classes. Another approach is to define the XML schemas for the DTOs, and then generate the Java classes from the schemas. **In this assignment, you should follow the latter approach.** You are given a schema for the Patient DTO, and an empty schema for the Treatment DTO. You will need to complete the schema for the Treatment DTO, and extend the schema with the specification for a Provider DTO. For simplicity with the JAXB Maven plugin, you can just put all the specifications in the one schema, although it is possible to have the Maven plugin generate JAXB classes for several schemas in a project.

The above example assumes that the different forms of treatments are defined using a choice element. If you want to use element substitution, then the above example would be recoded like this, where `JAXBElement` is used as a container since the actual root element depends on the form of the treatment:

```
package edu.stevens.cs548.clinic.service.dto.treatment;
```

```
public class TreatmentExporter
        implements ITreatmentExporter<JAXBElement<TreatmentType>> {
    private ObjectFactory factory = new ObjectFactory();
    public JAXBElement<TreatmentType> visitDrugTreatment
            (long tid, String diagnosis, String drug, int dosage) {
        DrugTreatmentType d = factory.createDrugTreatmentType();
        ...
        return factory.createDrugTreatment(d);
    }
    public JAXBElement<TreatmentType> visitSurgery (…) { ... }
    public JAXBElement<TreatmentType> visitRadiology(…) { ... }
}
```

Service Façades

You are provided with an Eclipse Maven EJB project. Use this to define patient and provider service façades for your SOA. The patient service façade provides operations for:

1. Adding a patient to a clinic. The operation takes a patient DTO and returns a new unique patient key. The operation should throw an exception if a patient with that patient identifier already exists in the database, or if the patient age provided is not consistent with the date of birth and current date.
2. Obtaining a single patient DTO, given a patient key (database primary key). An exception occurs for this operation if there is no patient for the specified patient key.
3. Obtaining a single patient DTO, given a patient identifier. An exception occurs for this operation if there is no patient for the specified patient identifier.
4. Return the DTO for a treatment, given a patient key and the key for the treatment. As a sanity check, the service should check that the treatment is in fact for the specified patient.

You are provided with a patient service with stubs for the operations. Complete this code to provide the above functionality. Most of the code just involves calling into the domain model.

The provider service façade provides operations for:

1. Adding a provider to a clinic. The operation takes a provider DTO as its argument.
2. Obtaining a single provider DTO, given a national provider identifier (NPI). An exception occurs for this operation if there is no provider for the specified provider identifier.
3. Obtaining a single provider DTO, given a provider key. An exception occurs for this operation if there is no provider for the specified key.
4. Adding a treatment for a patient. This operation takes a treatment DTO as its argument, that specifies the patient (by patient key) and the responsible provider (by provider key).
5. Return the DTO for a treatment, given a provider key and the key for the treatment. As a sanity check, the service should check that the treatment is in fact administered by the specified provider.

Session Beans

Define your service facades as stateless session beans (enterprise Java beans or EJBs). The application server will take care of instantiating an EJB from a pool of available instances for each client request that is made. Use the `@Stateless` annotation in Java EE to specify that the classes for your business logic are EJBs. Your session beans should offer both local and remote interfaces. For example, a possible deployment scenario would involve the Web tier and business tier in separate clusters. However you will just be deploying both tiers in the same JVM, so you will be using the local interfaces to access your services.

In order to retrieve PDOs from the database, your EJBs will have to use DAOs for the entity classes. You will use dependency injection to inject an entity manager into your service façade, and then instantiate a patient or provider DAO (as appropriate) with that entity manager. You will need a post-construct bean lifecycle callback to initialize the DAO, since dependency injection is performed after the constructor is executed. If you have followed the direction that you were provided with for the previous assignment, the patient DAO will ensure that each patient entity has a reference to a treatment DAO, as a transient field.

As with the previous assignment, provide a test bean that performs some testing of your application, writing the results of testing to the log. *Unlike the previous assignment, you should not access the domain model directly in the test bean. Instead, your test code will inject beans for the services that it needs, and invoke the logic of those service beans as POJOs (plain old Java objects).* Do this by modifying your start-up bean from the previous assignment (`InitBean`) to test the functionality of your service. This project should define a start-up bean for testing like this:

```
@Singleton
@LocalBean
@Startup
public class InitBean {

	private static Logger logger =
		Logger.getLogger(InitBean.class.getCanonicalName());

	private static void info(String m) {
		logger.info(m);
	}

	public TestService() {
	}

	@Inject
	private IPatientServiceLocal service;

	@PostConstruct
	public void init() {
		info("Initializing the service.");
```

```
            long patientA = service.addPatient(...);

            // Insert code to test the functions of your service.
            // Use info("...") to display the results in the log
        }
}
```

How should the dependency injection be done? One way is to use `@PersistenceContext` and `@EJB` annotations to inject entity managers and EJBs, respectively. However you will follow a better approach, that of *contexts and dependency injection* (CDI), which we will talk about more later in the course. In the service project, you will find a class called `ClinicDomainProducer`. This class shows how to define CDI for an entity manager. To complete this example, you should inject a value into the entity manager field in this producer class, using resource injection. Do this using `@PersistenceContext`, which should only be used in this producer class. (See the TODO item). *Note: It is important that you do not duplicate this class to other projects.*

Once you have done this, you can inject values into fields in your service objects as follows:

1. To inject an EJB with **local** interface `IFooLocal` into a field called `foo`, type: `@Inject IFooLocal foo;`
2. To inject an entity manager into a field called `em`, type `@Inject @ClinicDomain EntityManager em;`

We will explain this more fully when we discuss CDI later in the course. Note that this form of injection can only be done for local interfaces. In the sample code for the test bean above, a patient service bean is injected into the test bean, so that the business logic can be tested by invoking service operations. The domain model is not accessed directly in the test bean.

Since the ClinicInit project now depends on the ClinicService project (to obtain a reference to the entity manager), you will have to add a dependency on ClinicService in the ClinicInit POM file. See the dependency on ClinicDomain for an example of how to do this, or see the updated ClinicInit POM file that you are provided with.

Web Services

Your final task is to define Web services that provide access to your service facades outside your own middleware. There are two approaches to exporting a service facade as a Web service. One approach is to wrap your session bean as a Web service. Unfortunately the wizard in Eclipse for generating a Web service from a session bean does not support the Metro runtime that is used by Glassfish. The other approach is to deploy the service as a plain old Java object (POJO) in the Web tier, and this is the approach that you should use. In this approach, you define Web service interfaces and implementation classes, annotated with the `@WebService` annotation, in a dynamic Web

project (called `ClinicSOAPWebService`). The JSR-181 annotations for the Web service implementation should specify:

- Endpoint interfaces:
  `edu.stevens.cs548.clinic.service.web.soap.IPatientWebService` and
  `edu.stevens.cs548.clinic.service.web.soap.IProviderWebService`
- Target namespaces (for WSDL):
  `http://cs548.stevens.edu/clinic/service/web/soap/patient` and
  `http://cs548.stevens.edu/clinic/service/web/soap/provider`
- Service names (classes for the client API): `PatientWebService` and `ProviderWebService.`
- Service ports (classes for client proxies, obtained from the service class): `PatientWebPort` and `ProviderWebPort`

In the Web service interface, specify a SOAP binding for communication (document style, literal, wrapped). This is actually the default binding, but you should see how you might use JAX-WS annotations to customize the binding.

See e.g. this Web page for annotations you should use to specify your Web service:
`http://docs.oracle.com/cd/E13222_01/wls/docs103/webserv_ref/annotations.html#wp1105627`

You may also find this helpful:
`http://docs.oracle.com/cd/E13222_01/wls/docs103/webserv/jws.html#wp212031`

The compiled classes are bundled as a WAR (Web archive) file, and deployed in the Web tier. At run time, the Web service implementation obtains a reference to an EJB for the service façade using dependency injection (using the `@Inject` annotation), and delegates its service calls to this bean.

For testing purposes, add a `siteInfo()` method to the Web service interface (provided in the session bean), that returns a string value defined in the deployment descriptor for the session bean. The value returned by this service operation should be defined by resource injection from a deployment descriptor, not by hard-coding it into your code and not by using CDI.

Note: It is important that you choose different names for all of your Web service operations, since the types of the SOAP request messages are based on the names of the operations. Although overload resolution can resolve operations with the same name in Java, this does not help to distinguish SOAP request messages with the same name.

Submission

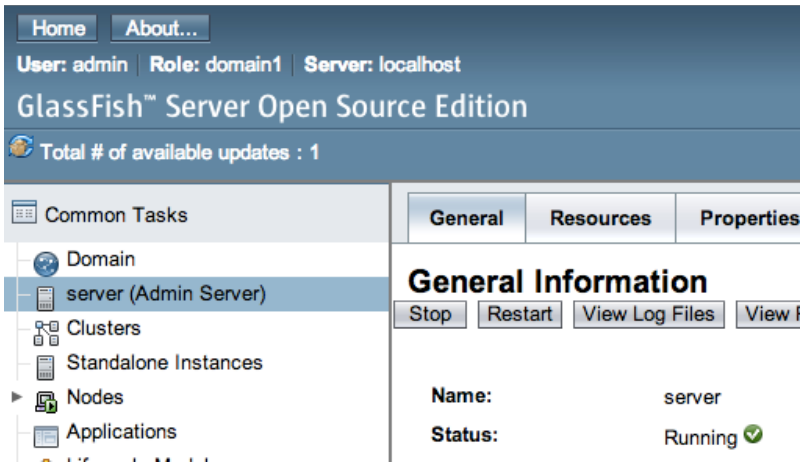
In addition to the classes from previous assignments (`ClinicDomain` and `ClinicInit`), this assignment requires the development of three additional projects: `ClinicDTOs` (for your DTO classes), `ClinicService` and `ClinicServiceClient` (for your EJBs) and `ClinicSOAPWebService` (a dynamic Web project that contains the definition of your SOAP Web service). You should add these additional projects to the deployment

assembly for your enterprise archive project, `ClinicApp`. In addition, you are provided with an update of the Maven parent module, `ClinicRoot`, that incorporates these additional Maven projects into the application. You should make sure that all of these projects are the same parent folder, otherwise the relative paths to and from the root project POM file will not work.

You should use `mvn clean install` to completely recompile your code, relying on Maven to handle dependencies between modules. You do not need to modify the Eclipse build path, because you are relying on Maven to handle dependencies. However, assuming the SOAP Web service project is created as a dynamic Web project in Eclipse, it has the project structure expected by Eclipse rather than that expected by Maven. Therefore you will have to follow this protocol to export the enterprise archive file:

- Clean the `ClinicApp` and `ClinicSOAPWebService` projects in Eclipse.
- Export the `ClinicApp` project as an enterprise archive file.

This will produce a file called `ClinicApp.ear`. Use the administration console in the application server to deploy your application, as you did with the previous assignment. The application server will instantiate the `InitBean` class to create a tester bean, and call its `init()` method once it is created. The tester should **not** access the domain directly, but should only call into the service logic (injected as EJBs into the tester bean). The output of the tests will be printed to the server log, which you can view from the Admin Server tab in the administration console (click on "`View Log Files`"):



The log viewer in Glassfish is buggy. If you are not seeing the logs that you expect, then you can view the log file in `/usr/local/glassfish4/glassfish/domains/domain1/logs/server.log` (using e.g. the vim editor).

Your solutions should be developed for Java EE 7. You should use Payara (Glassfish 4.1), Eclipse Oxygen, Java 8 and Eclipselink 2.5.x. In addition, record short mpeg or Quicktime videos of a demonstration of your assignment working (deploy the app and view the server log file in enough detail to see the output of your testing). Make sure that

your name appears at the beginning of the video. *Do not provide private information such as your email or cwid in the video*.

Your solution should be uploaded via the Canvas classroom. Your solution should consist of a zip archive with one folder, identified by your name. Within that folder, you should have at least these Eclipse projects: `ClinicRoot`, `ClinicApp`, `ClinicDomain`, `ClinicInit`, `ClinicDTOs`, `ClinicService` , `ClinicServiceClient` and `ClinicWebService`. You should also provide a report in the root folder, called `README.pdf`, that contains a report on your solution, as well as videos demonstrating the working of your assignment. Finally the root folder should contain the enterprise archive file (`ClinicApp.ear`) that you used to deploy your application.

You can deploy your enterprise application, including resource, application and presentation logic, as an enterprise archive file (EAR). Deploy this in the Glassfish application server, and save the WSDL file, including XML Schemas that are imported by the WSDL specifications, as part of your submission. You should include in your test videos one of you deploying the app and then navigating to the tester page that Glassfish provides to test your app. Test the "`siteInfo`" operation that returns the value injected from the deployment descriptor. The other service operations should be part of the test page, but you do not have to test the rest of the Web service API. The rest of your testing should be based on calling the service operations (invoked on the EJB interface) in the tester bean.

Your solution should be uploaded via the Canvas classroom, as a zip file. This zip file should have the same name as your Canvas userid. It should unzip to a folder with this same name, which should contain the files and subfolders with your submission.

It is important that you provide a document that documents your submission, included as a PDF document in your submission root folder. Name this document README.pdf. As part of your submission, export your Eclipse projects to your file system, and then include those folders as part of your archive file. Your written report should include, in an appendix, the XML schemas for your DTOs, as well as the WSDL for your Web services. It should also describe the important interfaces and APIs in your service-oriented architecture.