Module Development II

Adding services and data access objects

EHSDI

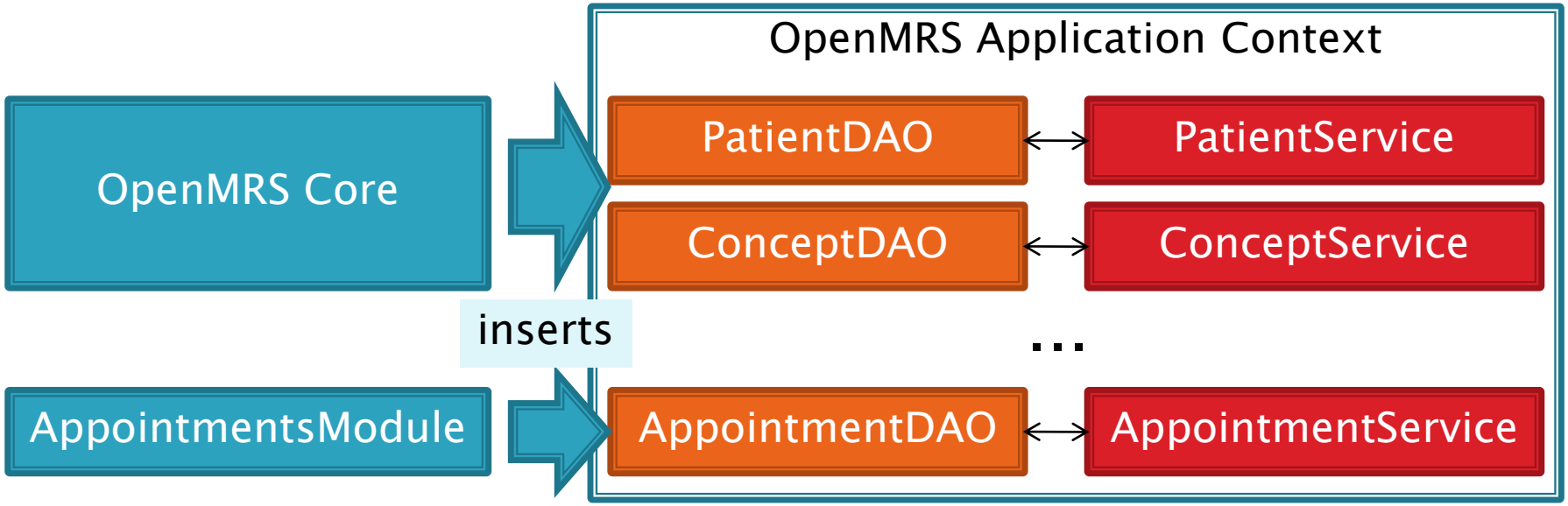
eBuzima

When and why?

- If a module is going to add to the OpenMRS data model then it's important to create DAOs and services
  - Is consistent with the programming model used by the rest of OpenMRS
  - Gives OpenMRS core and other modules access to the functionality of the module
- Even if a module doesn't have its own data, it might be appropriate to create a service for it
  - E.g. a module which analyzes existing data

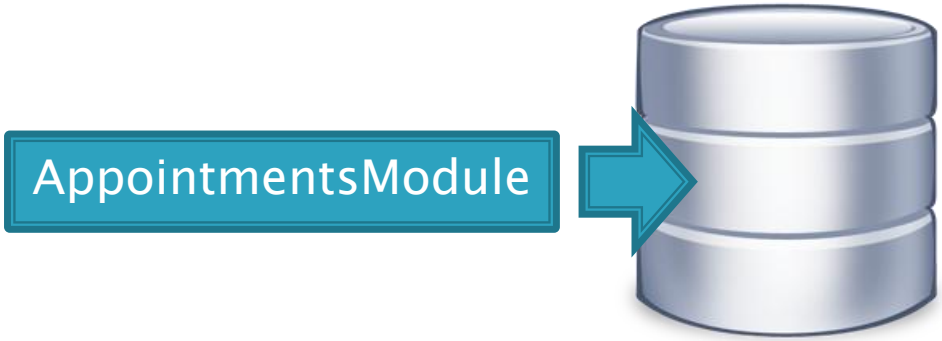
How?

- Modules can insert DAOs and services into the application context, in the same way that OpenMRS core does, e.g.



Adding to the database

- We could use our module activator class to execute SQL to add new tables, however...
  - We would need to check the tables don't already exist
  - What if we want to make schema changes to our module's database tables?



SqlDiff

- A mechanism in OpenMRS for a module to manage versions of its database tables
- A module includes a sqldiff.xml which tells OpenMRS...
  - How to create the tables for the module when its first installed
  - How to update the table schemas for newer versions of the module

Example (sqldiff.xml)

```
<sqldiff version="1.0">
  <diff>
    <version>1.0</version>
    <author>EHSDI</author>
    <date>Sept 30th 2010</date>
    <description></description>
    <sql>
      CREATE TABLE IF NOT EXISTS appointment (
        appointment_id int(11) NOT NULL auto_increment,
        patient_id int(11) default NULL,
        provider_id int(11) default NULL,
        location_id int(11) default NULL,
        date date default NULL,
        PRIMARY KEY  (appointment_id)
      ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
    </sql>
  </diff>
</sqldiff>
```

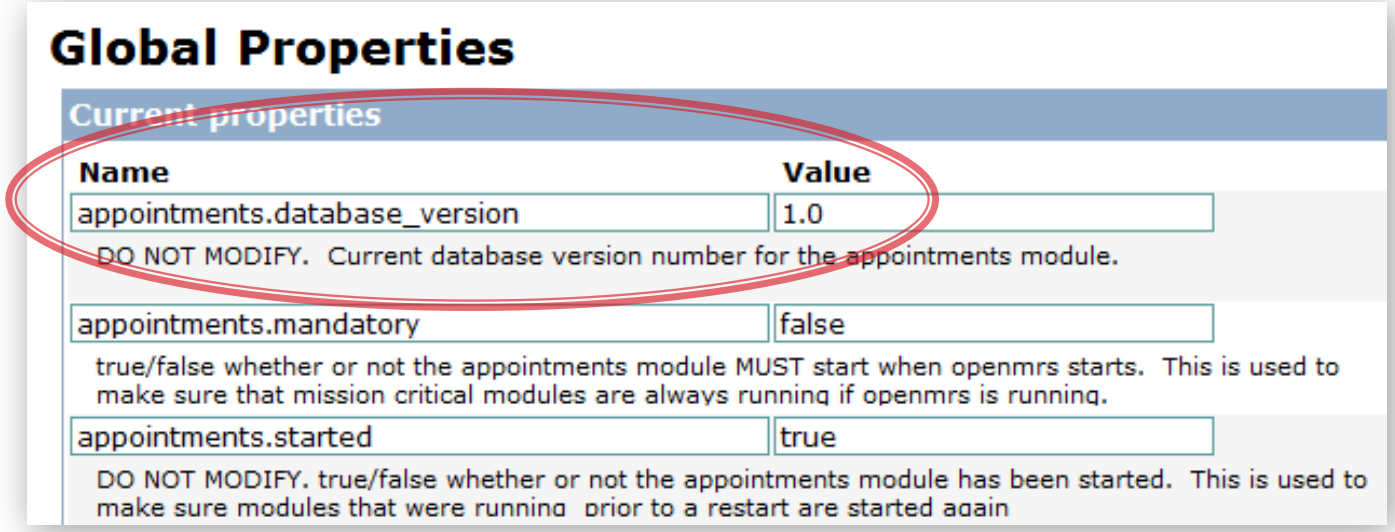
The version number of module's data model

Helpful information for the a developer

SQL to be executed

Example

- When we install the module, OpenMRS runs the `sqldif.xml` file, which
  - Creates the table
  - Creates a global property which records what version of the module's data model is in the database, i.e.



Example

- Now supposing we need to modify the data model. We need to...
  - Create a new version of the module's data model
  - Provide the SQL to get from the old version to the new
- We can do this by creating a new <diff>
- For example, if we want to add a TEXT field called *reason* to the appointments table...

Example (sqldiff.xml)

```
<sqldiff version="1.0">
  <diff>
    <version>1.0</version>
    <author>EHSDI</author>
    <date>Sept 30th 2010</date>
    ...
  </diff>

  <diff>
    <version>1.1</version>
    <author>EHSDI</author>
    <date>Sept 31st 2010</date>
    <description></description>
    <sql>
      ALTER TABLE appointment ADD reason TEXT DEFAULT NULL;
    </sql>
  </diff>
</sqldiff>
```

The old diff (1.0) remains unchanged

The new diff (1.1) makes the change to the schema

What happens...

- **If a user installs the module for the first time...**
  - They won't have a global property called `appointments.database_version`, so OpenMRS will consider itself to be at version 0
  - OpenMRS will run all the diffs whose version is greater than 0
  - This is will...
    - Create the table (1.0)
    - Add the new field to the existing table (1.1)

What happens...

- **If a user has version 1.0 of the module, and updates to version 1.1**
  - They will have a global property called `appointments.database_version` which tells OpenMRS that they have version 1.0
  - OpenMRS will run all the diffs whose version is greater than 1.0
  - This is will only...
    - Add the new field to the existing table (1.1)

Hibernate

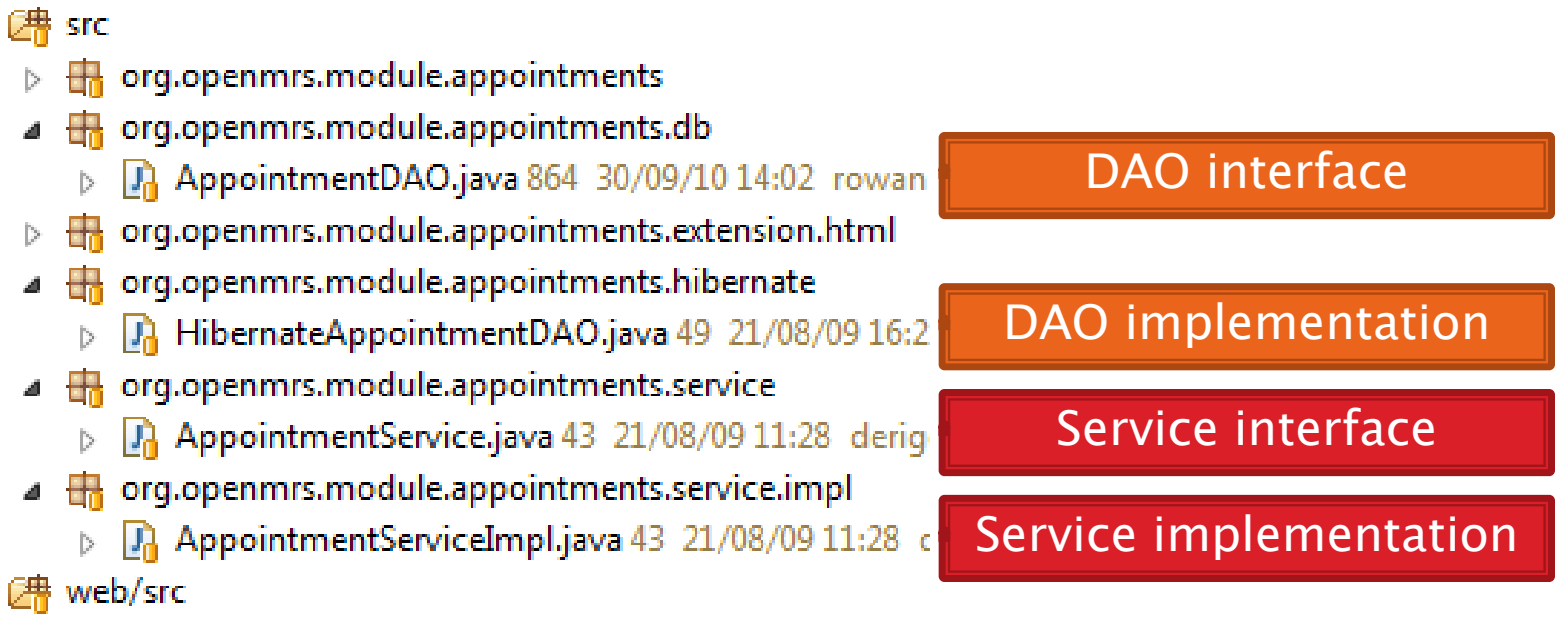
- We can also add Hibernate mapping files to map between our tables and our Java classes
- These are added to the *metadata* folder
- And then referenced in the module's `config.xml`, e.g.

```
<module configVersion="1.0">
  ...

  <mappingFiles>
    Appointment.hbm.xml
  </mappingFiles>
</module>
```

DAO and service

- DAOs and services are added to the module like any other Spring application, e.g.



DAO and service

- These then need to be added as beans like in a regular Spring application, e.g.

```
<bean id="appointmentDAO"
  class="org.openmrs.module.appointments.hibernate.HibernateAppointmentDAO">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>



<bean id="appointmentService"
  class="org.openmrs.module.appointments.service.impl.AppointmentServiceImpl">
  <property name="appointmentDAO" ref="appointmentDAO" />
</bean>
```

Defined in OpenMRS's application context

DAO and service

- Finally we need to register our service with OpenMRS so that it can be accessed like the other services, e.g.

```
ServiceContext.getInstance().getService(AppointmentService.class)
```

- This will...
  - Make it accessible from other modules or OpenMRS core
  - Allow it to use the transaction manager in OpenMRS core
  - Add support for the `@Authorized` annotation which is defined in OpenMRS

DAO and service

Calls `setModuleService()`

Class name to associate this service with

```
<bean parent="serviceContext">
  <property name="moduleService">
    <list>
      <value>org.openmrs.module.appointments.service.AppointmentService</value>

      <bean
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager" ref="transactionManager" />
        <property name="target" ref="appointmentService" />
        <property name="preInterceptors">
          <list>
            <ref bean="authorizationInterceptor" />
          </list>
        </property>
        <property name="transactionAttributeSource">
          <bean
 class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource" />
          </property>
      </bean>

    </list>
  </property>
</bean>
```

The proxy for the service that supports transaction management

References

- http://wiki.openmrs.org/display/docs/Creating+Modules