Providing your own data service implementation

In this session I will show how to implement the following Shindig services to connect your own datastore.

PersonService
ActivityService
AppDataService

During my first steps I have created another sample implementation where all the social data is held in memory. It's just another sample, but I think it is perfect for debugging reasons, because you can any time inspect your whole data at once.

Description

The full code is available as attachement (SampleShindigIntegrator.zip) to this site.

1. Make an implementation for all three interfaces

```java
public Future<Person> getPerson(UserId id, Set<String> fields, SecurityToken token) throws SocialSpiException {
    initDB();
    if (id != null) {
        Person currentPerson = db.findPerson(id.getUserId(token));
        if (currentPerson != null) {
            return ImmediateFuture.newInstance(currentPerson);
        }
    }
    throw new SocialSpiException(ResponseError.BAD_REQUEST, "Person not found");
}
```

Snippet of org.apache.shindig.social.spi.ShindigIntegratorPersonService.class

This implementation looks similiar to the sample implementation provided with Shindig. The big difference is, that the db instance is a instance of a class that holds the data in memory.
public class ShindigIntegratorPersistenceAdapter {

  private Map<String, Person> persons;
  ...
  private void init() {
    if (persons == null) {
      // create personlist
      persons = new HashMap<String, Person>();

      Name name = new NameImpl();
      name.setGivenName("James");
      name.setFamilyName("Bond");
      name.setFormatted("James Bond");
      Person john = new PersonImpl("john.doe", "007", name);
      ...
      persons.put(john.getId(), john);
    }
  }

  public Person findPerson(String id) {
    Person person = persons.get(id);
    return person;
  }
}
@Override
protected void configure() {
    bind(PersonService.class).to(ShindigIntegratorPersonService.class);
    bind(AppDataService.class).to(ShindigIntegratorAppDataService.class);
    bind(ActivityService.class).to(ShindigIntegratorActivityService.class);

    bind(ParameterFetcher.class).annotatedWith(Names.named("DataServiceServlet")).to(
        DataServiceServletFetcher.class);

    bind(Boolean.class).annotatedWith(Names.named(AnonymousAuthenticationHandler.ALLOW_UNAUTHENTICATED))
        .toInstance(Boolean.TRUE);

    bind(BeanConverter.class).annotatedWith(Names.named("shindig.bean.converter.xml")).to(
        BeanXStreamConverter.class);

    bind(BeanConverter.class).annotatedWith(Names.named("shindig.bean.converter.json")).to(
        BeanJsonConverter.class);

    bind(BeanConverter.class).annotatedWith(Names.named("shindig.bean.converter.atom")).to(
        BeanXStreamAtomConverter.class);

    bind(new TypeLiteral<List<AuthenticationHandler>>(){}).toProvider(
        AuthenticationHandlerProvider.class);
}

Snippet of org.apache.shindig.social.core.config.ShindigIntegratorGuiceModule.class

This class binds the Shindig services to the new implementations. So they will be called instead of the sample coming with Shindig. Another way to tell Shindig which implementation to use is to change the @ImplementedBy annotation in for example PersonService.class. But to me the binding solution sounds better.

For unit tests I have registered the classes in a guice module specifically for unit tests. This is done the same way as in the upper guice module. In the unit test itself this modul has to be called on setup. In the test methods the service can then be used directly.
@Before
public void setUp() throws Exception {
    Injector injector = Guice.createInjector(new
    ShindigIntegratorTestsGuiceModule());
    personService =
    injector.getInstance(ShindigIntegratorPersonService.class);
}
.
.
.
/**
 * Tests getPerson with a UserId typed as Userid. The Id string is set in
the UserId.
 * @throws Exception
 */
@Test
public void testGetExpectedPersonByUserId() throws Exception {
    Future<Person> selectedObject = personService.getPerson(JOHN_DOE,
    Collections.<String> emptySet(),
    new FakeGadgetToken());
    Person person = selectedObject.get();
    assertEquals(JOHN_DOE.getUserId(), person.getId());
}

Snippet of ShindigIntegratorPersonServiceTest.class

3. Register the guice module in the web.xml

In the web.xml the param guice-modules has to be changed to reference the new implementation.

    <context-param>
    <param-name>guice-modules</param-name>
    <param-value>
        org.apache.shindig.common.PropertiesModule:
        org.apache.shindig.gadgets.DefaultGuiceModule:
        org.apache.shindig.social.core.config.ShindigIntegratorGuiceModule:
        
        org.apache.shindig.gadgets.oauth.OAuthModule:
        org.apache.shindig.common.cache.ehcache.EhCacheModule
    </param-value>
    </context-param>

Snippet of web.xml

There are several web.xml files in the shindig installation. Comments in the web.xml files indicate what they're for:

- web.gadgets.xml: just the gadget rendering server
- web.social.xml: just the social data / RESTful server
- web.full.xml: both servers in a single container

4. Rebuild Shindig and you are ready

After all maven has to be run with a clean install and things are done.
After this you can run the samplecontainer as before. Just things like a dump in the samplecontainer page doesn't work as I did not implement it.

**Conclusion**

This documentation shows how to implement the interfaces for Shindig. Next step would be to connect to a real database.

I think Implementing this services is very straightforward. The hardest thing is to get familiarized with Shindig.