This page last changed on Mar 13, 2011 by patrik_nordwall.

Sculptor App Engine Tutorial

Sculptor provides an implementation for Google App Engine (below referred to as GAE).

In this tutorial we will show you how you can create a project, generate a complete persistence and service layer. Create a simple Springframework MVC client. How to unit test it. How to test it your local GAE environment. And finally, how to deploy it to GAE. We assume you have your Sculptor environment installed. And we are going to use GAE Eclipse plugin, so if you haven't installed it yet, do so from google. Also, we are going to show how to deploy to GAE. For you to do this, you need an GAE account. You can sign up here if you doesn't have one.

Quick start

A fast track to a deployment in the cloud.

Setup project

First we will setup the project structure for maven and eclipse.

1. Use the following command (one line) to create a maven pom and file structure. You can change the groupId and artifactId if you like.

mvn archetype:generate -DarchetypeGroupId=org.fornax.cartridges -DarchetypeArtifactId=fornax-cartridges-sculptor-archetype-appengine -DarchetypeVersion=2.0.0 -DarchetypeRepository=http://fornax-platform.org/nexus/content/repositories/public

Fill in groupId and artifactId:

```
Define value for groupId: : org.helloworld
Define value for artifactId: : hellogae
Define value for version:  1.0-SNAPSHOT: :
Define value for package:  org.helloworld: :
```

Ignore warnings
There will be warnings like this:

```
[WARNING] org.apache.velocity.runtime.exception.ReferenceException: reference :
template = archetype-resources/pom.xml [line 40,column 42] : ${fornax-oaw-m2.ver
sion} is not a valid reference.
```

Ignore these warnings and continue with next step if you see no errors.

2. In the new directory, run

```
mvn clean
```

```
mvn generate-sources
```

```
mvn eclipse:eclipse
```

to generate example sources and create an Eclipse project with the same dependencies as in the pom.

3. Open Eclipse and import the project.

Run in local GAE

Since Sculptors maven-gae-archetype provides you with a simple example application you are now ready to test it.

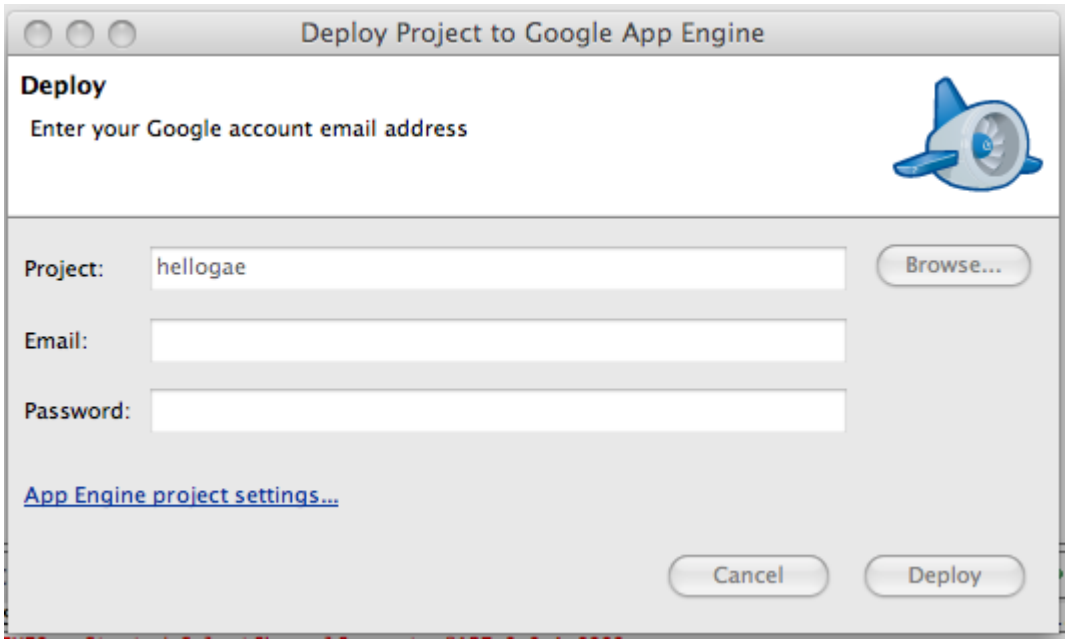
The GAE Eclipse plugin comes with a local environment. Lets try to run our new application in it.

Open the 'Run As' menu for your project and choose to run it as 'Web Application'.

When you see the message: 'The server is running at http://localhost:8888/'
point you favorit browser to that address and you should see something like this:

Deploy to GAE

To deploy to appengine its just a matter of pushing the right button in the toolbar: 
Add the prompted information and push the 'Deploy' button:



If you click the 'App Engine project settings...' link in the window, you can set the application id. This is the id of one of your applications that you have attached to your GAE account.

And after half a minute (or whatever your connection speed allows) you should be able to browse the same application but now its on the cloud

Sculptor Archetype Appengine

So, perhaps it seemed like magic when creating the project. But actually its not. What you get when running the archetype is:

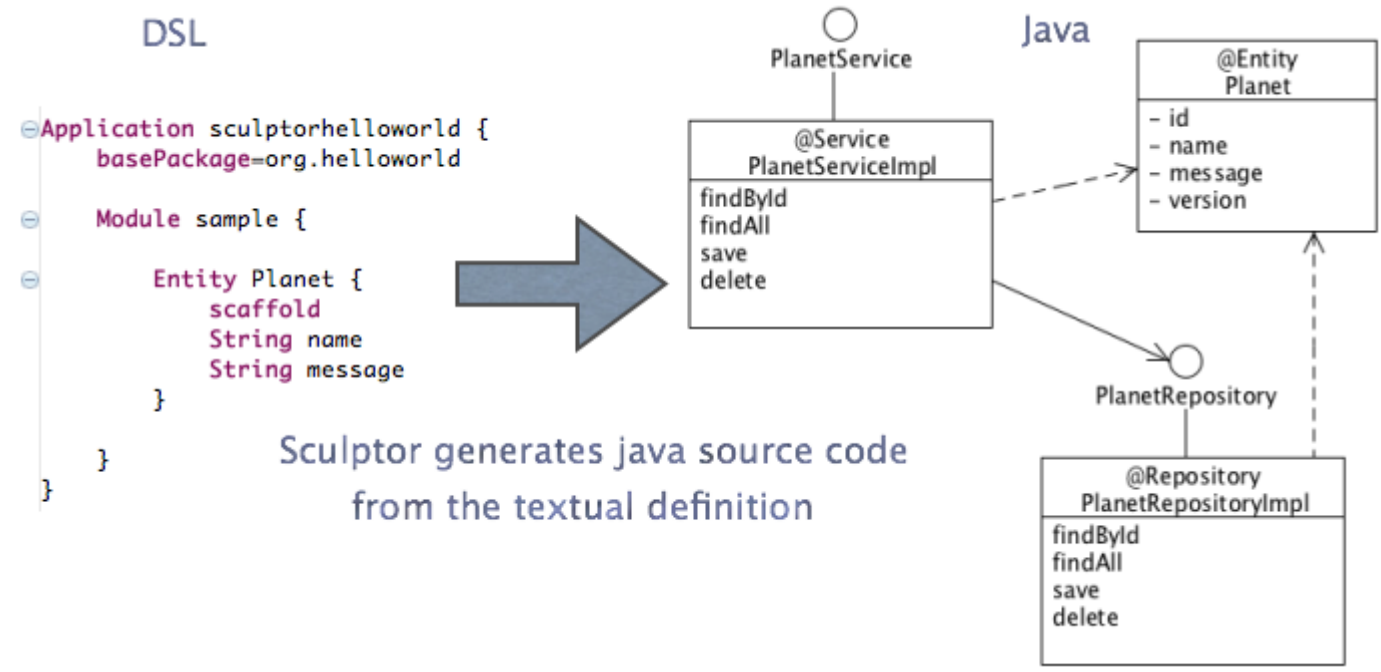
pom.xml

A pre configured pom-file with all the right dependencies, repositories and hooks into the maven build life cycle that copies the dependencies to the right folder to make the GAE tools pick them up when doing deploy's etc.

model.btdesign

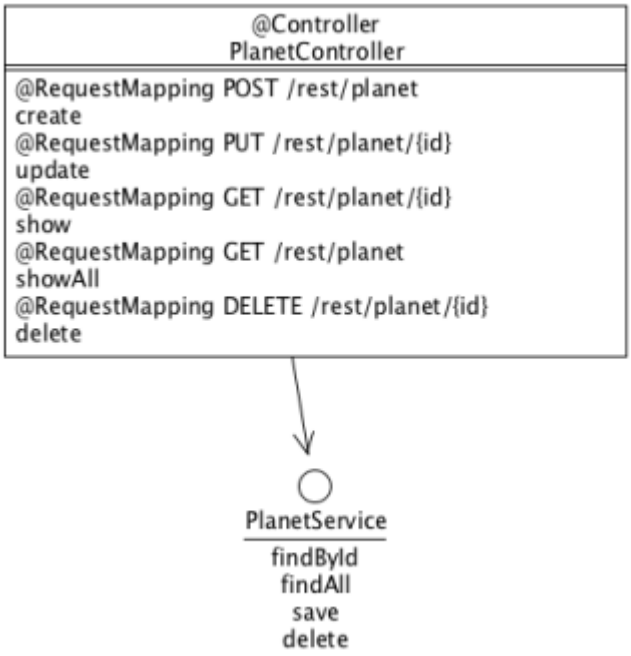
The archetype creates a simple sample model, from which Sculptor generates Entity, Repository and Service with the default CRUD operations; findById, findAll, save, and delete.
The model is defined in a textual DSL, with an intuitive syntax, from which Sculptor generates high quality Java code and configuration. It is not a one time shot. The application can be developed

incrementally with an efficient round trip loop. The generator is part of the build process (maven).



Spring MVC files

The archetype creates a sample of a Spring Controller and JSP pages for the CRUD operations. The set of files works together with the above Planet model.



It is possible and encouraged to use the REST support in Sculptor for GAE applications.

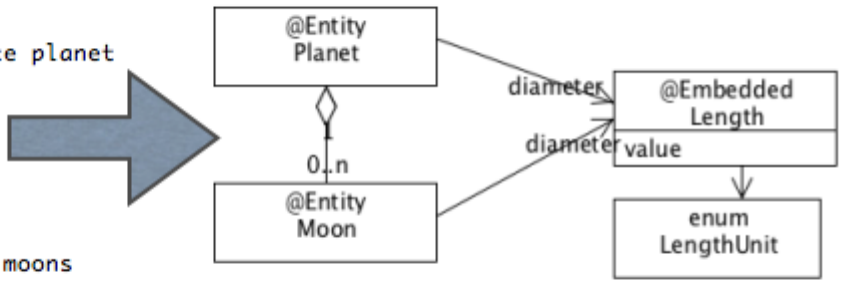
One module
As of now it is a one module solution. This means that both business tier and client tier resides in the same module.

Relations are special

Since the persistence mechanism behind GAE is BigTable and not an ordinary relational database, relations are very limited.
Sculptor generates JPA mapping annotations for the domain objects defined in the design model.

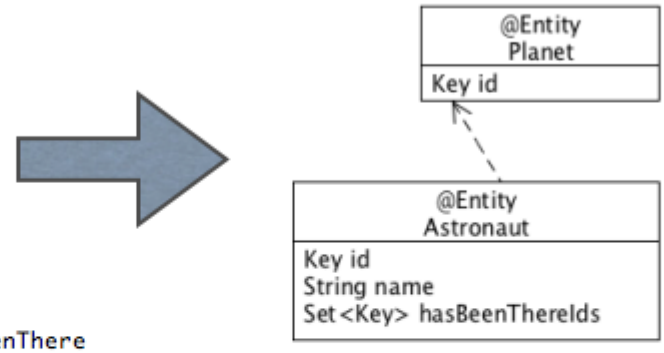
Owned and embedded associations are supported and mapped as ordinary JPA associations. They are specified with aggregate and BasicType in the Sculptor model.

```
Application sculptorhelloworld {
    basePackage=org.helloworld

    Module sample {

        Entity Planet {
            scaffold
            String name
            String message
            - @Length diameter
            - Set<@Moon> moons opposite planet
        }

        Entity Moon {
            not aggregateRoot
            String name
            - @Length diameter
            - @Planet planet opposite moons
        }

        BasicType Length {
            BigDecimal value
            - @LengthUnit unit
        }

        enum LengthUnit {
            m, km, miles
        }
```



Unowned associations are handled with id references and you must lookup the objects with findById when needed.

```
Application sculptorhelloworld {
    basePackage=org.helloworld

    Module sample {

        Entity Planet {
            String name
            String message
        }

        Entity Astronaut {
            String name
            - Set<@Planet> hasBeenThere
        }
    }
}
```




```
@Column(name = "hasBeenThere", nullable = false)
private Set<Key> hasBeenThereIds = new HashSet<Key>()

public Set<Key> getHasBeenThereIds() {
    // appengine sometimes stores the collection as n
    if (hasBeenThereIds == null) {
        hasBeenThereIds = new HashSet<Key>();
    }
    return hasBeenThereIds;
}
```

For more information on how relations are handled, see google's doc.

JUnit testing

Sculptor makes it easy to write JUnit tests for Google App Engine. A test case looks like this:

```
public class PlanetServiceTest extends AbstractAppEngineJpaTests {
    @Autowired
    private PlanetService planetService;
    @Before
    public void populateDatastore() {
        Planet earth = new Planet("Earth");
        getEntityManager().persist(earth);
        Planet mars = new Mars("Mars");
        getEntityManager().persist(mars);
    }
    @Test
    public void testFindAll() throws Exception {
        List<Planet> all = planetService.findAll(getServiceContext());
        assertEquals(2, all.size());
    }

    @Test
    public void testFindByName() throws Exception {
        Planet found = planetService.findByName(getServiceContext(), "Mars");
        assertNotNull(found);
        assertEquals("Mars", found.getName());
    }
}
```

Very natural!

It is interesting to take a look at the base class. It defines a few annotations and extends AbstractJUnit4SpringContextTests to initialize the Spring environment. This enables usage of ordinary @Autowire dependency injection directly in the test class.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations =  {"classpath:applicationContext-test.xml"})
public abstract class AbstractAppEngineJpaTests extends AbstractJUnit4SpringContextTests {
```

The embedded App Engine environment is initialized from a method annotated with @Before, i.e. invoked before each test method.

```
public static void setUpAppEngine(ApiProxy.Environment testEnvironment) {
    ApiProxy.setEnvironmentForCurrentThread(testEnvironment);
    ApiProxy.setDelegate(new ApiProxyLocalImpl(new File(".")) {
    });
    ApiProxyLocalImpl proxy = (ApiProxyLocalImpl) ApiProxy.getDelegate();
    proxy.setProperty(LocalDatastoreService.NO_STORAGE_PROPERTY, Boolean.TRUE.toString());
    clearSentEmailMessages();
}

public static void tearDownAppEngine() {
    ApiProxyLocalImpl proxy = (ApiProxyLocalImpl) ApiProxy.getDelegate();
    LocalDatastoreService datastoreService = (LocalDatastoreService) proxy.getService("datastore_v3");
    datastoreService.clearProfiles();
    clearSentEmailMessages();
}
```

It is initialized with in memory data store, i.e. it is empty before each test method. You may populate it with initial data in your subclass in a @Before method, see populateDataStore in the sample above.

Transactional gotchas
When working with ordinary databases the Spring transactional test support is very useful, i.e. Spring executes each test method in a transaction, which is rolled back after the test method. That is achieved with the following annotations and usage of the annotation @BeforeTransaction instead of the ordinary @Before.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations =  {"classpath:applicationContext-test.xml"})
@TestExecutionListeners(TransactionalTestExecutionListener.class)
@TransactionConfiguration(transactionManager = "txManager", defaultRollback = true)
@Transactional
public abstract class AbstractAppEngineJpaTests extends AbstractJUnit4SpringContextTests {
```

This approach doesn't work when doing appengine tests. It will fail on the last assert when using the above transactional support.

```
@Test
public void testSave() throws Exception {
    int countBefore = countRowsInTable(Planet.class);
    Planet jupiter = new Planet("Jupiter");
    supplierService.save(getServiceContext(), jupiter);
    int countAfter = countRowsInTable(Planet.class);
    assertEquals(countBefore + 1, countAfter);
```

```
}
```

The reason is that queries see a snapshot of the datastore as of the beginning of the transaction.

Data isolation between test methods is no problem, since the datastore is initialized (empty) before each test method.