Objectify: Object Relational Mapping for Google AppEngine

Executive Summary

The objectify-appengine library (objectify for short) makes using the Google AppEngine datastore much easier. Specifically, it allows developers to add objects directly to the datastore, and to retrieve objects from the datastore by specifying their class and, optionally, field values. In this lab you will recreate the functionality of the AppEngine lab from last week, and see how much simpler objectify-appengine makes it.

This tutorial is based on materials from Dr. Christine Julien, Dr. Adnan Aziz and Dr. Miryung Kim.

Installing Objectify

The following screenshots and procedures were developed using the Juno Eclipse release (version 4.2).

Step 1: Software required

This lab requires an Eclipse installation with Google Plugin for Eclipse for App Engine

This lab requires the library objectify: http://repo1.maven.org/maven2/com/googlecode/objectify/objectify/4.0b1/objectify-4.0b1.jar

Step 2: Add objectify.jar to your project

Add objectify-4.0b1.jar to your project's WEB-INF/lib directory. There are no other jar dependencies.

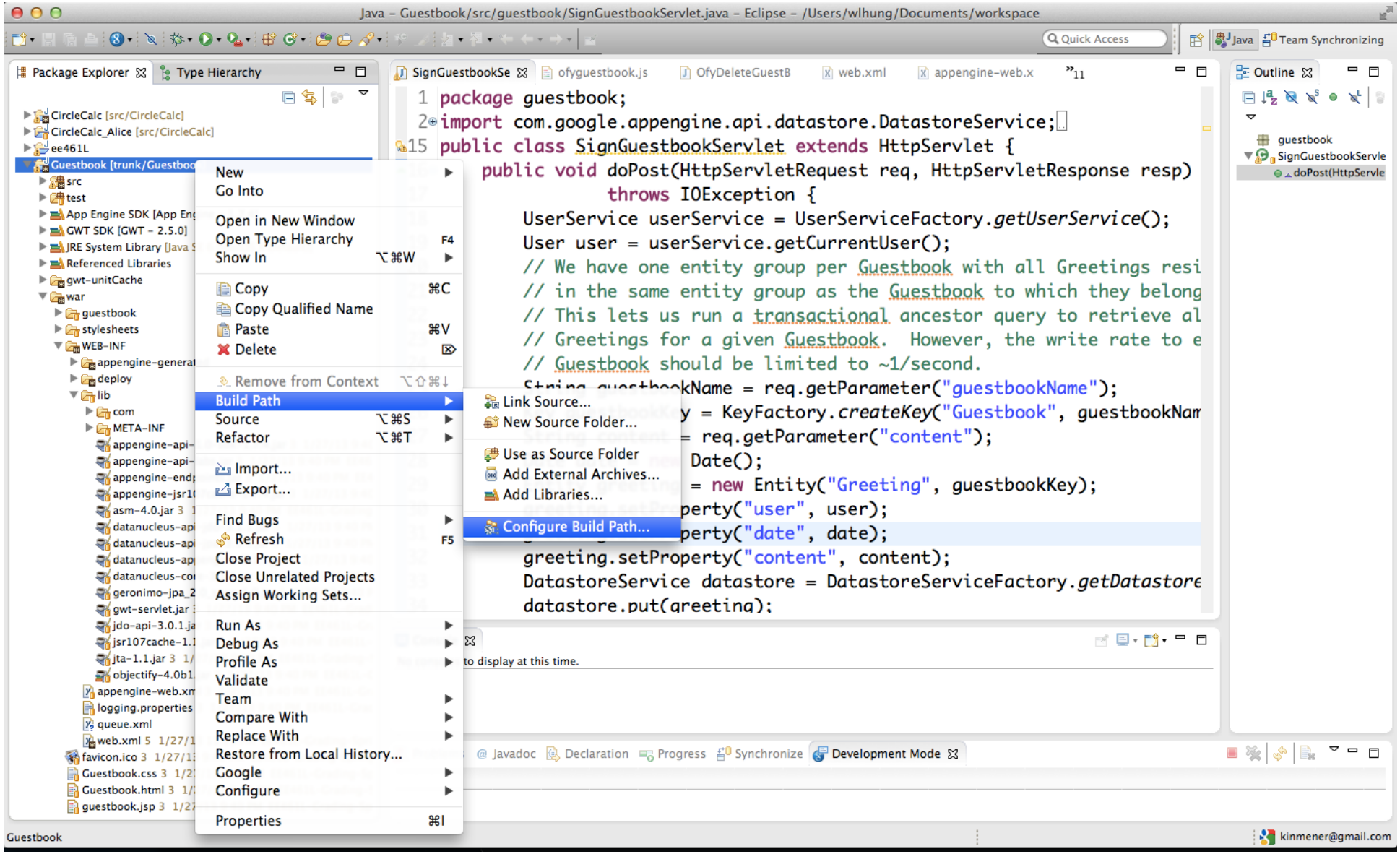
Step 3: Enable ObjectifyFilter for your requests https://github.com/objectify/objectify

Objectify requires a filter to clean up any thread-local transaction contexts and pending asynchronous operations that remain at the end of a request. Add this to your WEB-INF/web.xml:

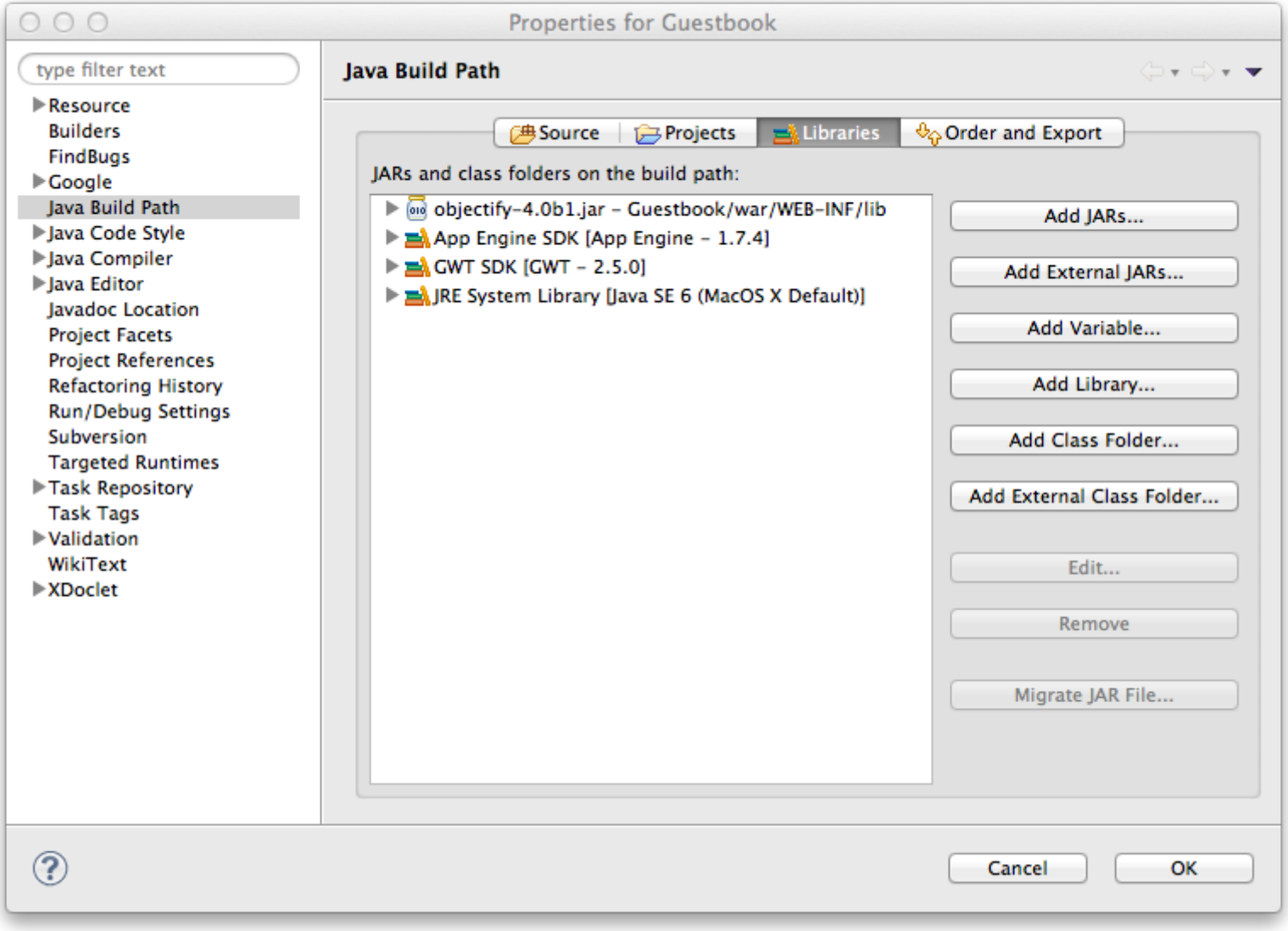
```
<filter>
	<filter-name>ObjectifyFilter</filter-name>
	<filter-class>com.googlecode.objectify.ObjectifyFilter</filter-class>
</filter>
<filter-mapping>
	<filter-name>ObjectifyFilter</filter-name>
	<url-pattern>/*</url-pattern>
</filter-mapping>
```

Step 4: Configure Build Path

Right click on your project and select build path -> configure build path



Click “Libraries” -> “Add JARs” to add your objectify-4.0b1.jar



Using Objectify

Step 1: Entities

To use Objectify, you first need to define the Entities that will be part of your application. You do this using Objectify's annotations. Let's look at some general rules first, then we will do a full example Greeting class for your Guestbook application.

- Entity classes must be annotated with @Entity.

- Objectify persists fields and only fields. It does not arbitrarily map fields to the datastore; if you want to change the way a field is stored, rename the field. Getters and setters are ignored, so you can isolate the public interface of your class. That is, you can still define getters and setters for your Entity (and should) and you can use your Java implementation to interact with your Entity using these methods.
- Objectify will not persist static fields, final fields, or fields annotated with @Ignore. It will persist fields with the transient keyword, which only affects serialization.
- Entities must have one field annotated with @Id. The actual name of the field is irrelevant, and the field can be renamed at any time, even after data is persisted. This value becomes part of the Key which identifies an entity.
- The @Id field can be of type Long, long, or String. If you use Long and save an entity with a null id, a numeric value will be generated for you using the standard GAE allocator for this kind. If you use String or the primitive long type, values will never be autogenerated.
- You can persist any of the core value types, Collections (e.g., Lists and Sets) of the core value types, or arrays of the core value types. You can also persist properties of type Key (the native datastore type) and Key<?>) (the Objectify generified version). There are more advanced types that can be persisted, which will be discussed later.
- There must be a no-arg constructor (or no constructors - Java creates a default no-arg constructor). The no-arg constructor can have any protection level (private, public, etc). Do you know why you might want a private constructor?
- String fields that store more than 500 characters (the GAE limit) are automatically converted to Text internally. Note that Text fields, like Blob fields, are never indexed (see Queries).
- byte[] fields are automatically converted to Blob fields internally. However, Byte[] is persisted "normally" as an array of (potentially indexed) Byte objects. Note that GAE natively stores all integer values as 64-bit long values.

To do all of this for our Guestbook application, we will create a Greeting class under the guestbook directory that looks like the following:

```
package guestbook;

import java.util.Date;


import com.google.appengine.api.users.User;

import com.googlecode.objectify.annotation.Entity;

import com.googlecode.objectify.annotation.Id;



@Entity

public class Greeting implements Comparable<Greeting> {
```

```
    @Id Long id;
    User user;
    String content;
    Date date;
    private Greeting() {}
    public Greeting(User user, String content) {
        this.user = user;
        this.content = content;
        date = new Date();
    }
    public User getUser() {
        return user;
    }
    public String getContent() {
        return content;
    }
    @Override
    public int compareTo(Greeting other) {
        if (date.after(other.date)) {
            return 1;
        } else if (date.before(other.date)) {
            return -1;
        }
        return 0;
    }
}
```

Don't just copy this over to your workspace, but really go through it and make sure you understand what's being done.

Step 2: Register Entities

Before you use Objectify to load or save data, you must register all entity classes for your application. Objectify will introspect these classes and their annotations to build a metamodel, which is used to efficiently manipulate entities at runtime.

- Registration must be done at application startup, before Objectify is used.
- Registration must be single-threaded. Do not register() from multiple threads.
- All entity classes must be registered, including polymorphic subclasses.
- @Embed classes do not need to be registered.

We recommend that you register classes in a static initializer for something that is used at application startup. For example:

```
static {
      ObjectifyService.register(Greeting.class);
  }
```

You're going to want to do this in your servlet class that is loaded at startup. We're going to make a new servlet for the Guestbook in a minute, so sit tight.

Step 3: Basic Objectify Operations

[Note: We're just reading and learning right now; we'll come back to the "doing" in a little bit.]

The Objectify Instance

All datastore operations begin with an instance of Objectify. An Objectify instance holds the complete state of your persistence session:

- A transaction context, if present
- A cache of entity objects loaded in the current session
- Parameters for deadline, consistency setting, and caching

An Objectify instance should be used from a single thread only. It is a cheap throwaway object that represents a session of activity. Typically you will re-use a single instance through a request, except when you begin transactions.

Obtaining an Objectify Instance

The preferred way to obtain an Objectify instance is to call a static method:

```
Thing th = ObjectifyService.ofy().load().key(thingKey).get();
```

To keep the typing to a minimum, we recommend a static import:

```
import static com.googlecode.objectify.ObjectifyService.ofy;
Thing th = ofy().load().key(thingKey).get();
```

ObjectifyService.ofy() will always return the correct instance for your thread and transaction context and provides you with an "always available" persistence context without having to pass extra parameters to your business methods. The instance returned by ofy() will change when you enter and exit transactions. Be sure to install the ObjectifyFilter so that the last instance is cleaned up at the end of a request.

As a general practice, we recommend not holding on to Objectify instance variables:

```
// This is generally a bad practice:
Objectify ofy = ofy();
ofy.load()...
ofy.save()...

// This is safer:
ofy().load()...
ofy().save()...
```

Consistently using ofy() eliminates potential confusion when working with multiple layers of transaction depth. This is discussed in more detail in Transactions, though this tutorial skips that for now.

Loading

Objectify provides a variety of ways to load entities from the datastore. They all begin with a command chain started by load(). Note that all operations are asynchronous until you manifest an actual entity POJO (a “plain old java object”); all the Map, List,

and Ref interfaces hide the asynchrony.

This covers the type of requests that all translate into a get-by-key in the datastore. For another way to load data, see Queries.

```
// Simple key fetch, always asynchronous
Ref<Thing> th = ofy().load().key(thingKey);

// Usually you don't need async, so call through the Ref<?>
Thing th =  ofy().load().key(thingKey).get();         // return null if not found
Thing th =  ofy().load().key(thingKey).safeGet();     // throws NotFoundException

Ref<Thing> th = ofy().load().entity(entity);  // Equivalent to ofy().load().key(get_key_of(entity));
Ref<Thing> th = ofy().load().value(rawKey);   // Accepts anything key-like; Key, Key<?>, Ref<?>, etc.

// Multi get is async, hides asynchrony behind Map interface
Map<Key<Thing>, Thing> ths = ofy().load().keys(thingKey1, thingKey2, thingKey3);
Map<Key<Thing>, Thing> ths = ofy().load().keys(iterableOfKeys);
// There are also entities() and values() methods

// Fetching by id - this is really just syntactic sugar for get-by-key
Ref<Thing> th = ofy().load().type(Thing.class).id(123L);
Thing th =      ofy().load().type(Thing.class).id(123L).get();

// With a parent
Ref<Thing> th = ofy().load().type(Thing.class).parent(par).id(123L);

// Batch get, asynchrony is hidden behind Map
Map<Key<Thing>, Thing> ths = ofy().load().type(Thing.class).ids(123L, 456L, 789L);
```

Load Groups

When you use fields to establish relationships between entities, you can use the @Load annotation to automatically retrieve an object graph using the optimal number of batch fetches. However, you do not always want to load an entire graph.

Objectify provides load groups, inspired by Jackson's Json Views. Here is the simplest usage:

```
@Entity
public class Thing {
    public static class Everything {}

    @Id Long id;
    @Load(Everything.class) Ref<Other> other;
}

// Doesn't load other
Thing th = ofy().load().key(thingKey).get();

// Does load other
Thing th = ofy().load().group(Everything.class).key(thingKey).get();
```

Saving

The following shows the API of objectify to save entities

```
ofy().save().entity(thing1);         // asynchronous
ofy().save().entity(thing1).now();   // synchronous

ofy().save().entities(thing1, thing2, thing3);        // asynchronous
ofy().save().entities(thing1, thing2, thing3).now();  // synchronous
```

```
List<Thing> things = ...
ofy().save().entities(things);       // asynchronous
ofy().save().entities(things).now();  // synchronous
```

Objectify Your Guestbook

So you've made your Greeting class (above). Now we need to create a Guestbook form that uses Objectify, and a servlet that uses Objectify.

Step 1: The Guestbook form with Objectify

Create a copy of guestbook.jsp called ofyguestbook.jsp in /war. You're going to remove all of the references to the Datastore. We're going to use the java.util.Collections API and the com.googlecode.objectify.Objectify and com.googlecode.objectify.ObjectifyService classes; add those to the import statements at the top of the form. (**Edit: What works for many is to import com.googlecode.objectify.***)

We're still going to use the UserService in the same way, and the form's presentation can state the same. But scroll down to where you were getting access to the Datastore previously. Here, we want to interact with Objectify instead. Instead of running the query and listing the retrieved objects as we did before, we want to do the following (register the Greeting Entity, grab the greetings, then sort them).

```
ObjectifyService.register(Greeting.class);
List<Greeting> greetings = ObjectifyService.ofy().load().type(Greeting.class).list();
Collections.sort(greetings);
```

Here's a question: what does Collections.sort(greetings); do? Why? Did you implement that?

Now, instead of having a List of Entity types like we had before, we have a List of Greeting types. Update your code to reflect that. You'll have to change both the type statements but also how you access fields of that type. We won't use the "getProperty" method anymore; instead our Greeting class has getter and setter methods.

One last thing: we're going to differentiate all of our forms and servlets from our old non-objectified version of the Guestbook. So at the bottom, when it defines the form, instead of the "action" being "/sign", it should be "/ofysign".

Step 2: The Guestbook Servlet with Objectify

The next step is to make a servlet that uses Objectify. Create a copy of SignGuestbookServlet.java called OfySignGuestbookServlet.java. You can remove all of the imports about the Datastore. We need to import com.googlecode.objectify.ObjectifyService. We're also going to want to do the static import of ofy as described above:

```
import static com.googlecode.objectify.ObjectifyService.ofy;
```

Next, we need to register our Greeting class (use the static block approach to initializing a field of the servlet; see above).

We're going to ignore the name of the guestbook, so you can delete the stuff in doPost() that relates to that. You can also delete the stuff that relates to the Datastore since we're going to replace that with Objectify. All we really need to do in doPost() now is the following:

- Access the User object from the UserService.
- Get the content from the request.
- Create a new Greeting using the user and the content.
- Chuck the Greeting into Objectify using a synchronous call (see above).
- Send the response (be sure to redirect the user to the ofyguestbook.jsp that you defined instead of guestbook.jsp).

Step 3: Update web.xml

The last piece is to update web.xml to announce our new servlet (create a new <servlet> and <servlet-mapping>). You can also update the welcome-file if you want. Right now, it points to guestbook.jsp. If you don't change it and want to test your new ofyguestbook.jsp, you'll have to explicitly enter that url, since guestbook.jsp will remain the default.

What to Submit

Submit the source for your new servlet (e.g., I called mine OfySignGuestbookServlet.java). In the first line of the file, include a comment that has the URL for your working Objectified app. Please submit something of the form "http://<app-id>.appspot.com/ofyguestbook.jsp" that points explicitly to your new jsp.

Note: If you already submitted both a website URL and a source file, don't worry, that's fine.