Implementing OAuth in HPI IP

Richard Metzler

Hasso-Plattner-Institut
Universität Potsdam
http://www.hpi.uni-potsdam.de/

Abstract. Today people are using many different internet services per day. It is hard for users to remember different usernames and passwords for every service they use so they often reuse the same username and password on different sites. This is a security issue that protocols like OpenID and OAuth try to solve. By creating a single online identity and allowing third party sites to authenticate users an identity provider is able to solve the security issue and enable single sign-on (SSO) above different websites.
By adding an OAuth service provider to the existing HPI Identity Provider we enable third party services to authorize users via OAuth and manage the user's online identities.

1 Introduction

Every day people use many different internet websites for their online activities. While some of these websites are used by only a few to several thousend people, some other websites like webbased email programms or social networking websites are visited by millions of users on a daily basis. Google's GMail online email application, Facebook and Twitter aim to become the center of our online activities and therefor the single provider of our online identity. One way for them to achieve this is to enable 3rd party sites to allow users to sign up and log in with accounts from the identity provider. This is possible through online identity protocols like OpenID and OAuth that are able to authenticate a user and redirect him back to the third party website.

For users this feature enables a more secure and better online experience as they are not required to register with username and password at every website they want to try out. For the third party service this often means that their sign-up conversion rate can dramatically increase.

In this paper we describe the OAuth 1.0a protocol. Then we describe our proposed privacy service that is granted access to the HPI identity provider via OAuth. We explain the changes we made in order to enable OAuth in the HPI-IP and how the API works.

1.1 The OAuth Protocol

OAuth is an open protocol to allow secure API authorization in a simple and standard method from desktop and web applications. [OAuth-spec] It enables

users to authenticate and authorize 3rd party applications called **OAuth consumers** to access data that is associated with a **restricted resource** managed by the **service provider**.

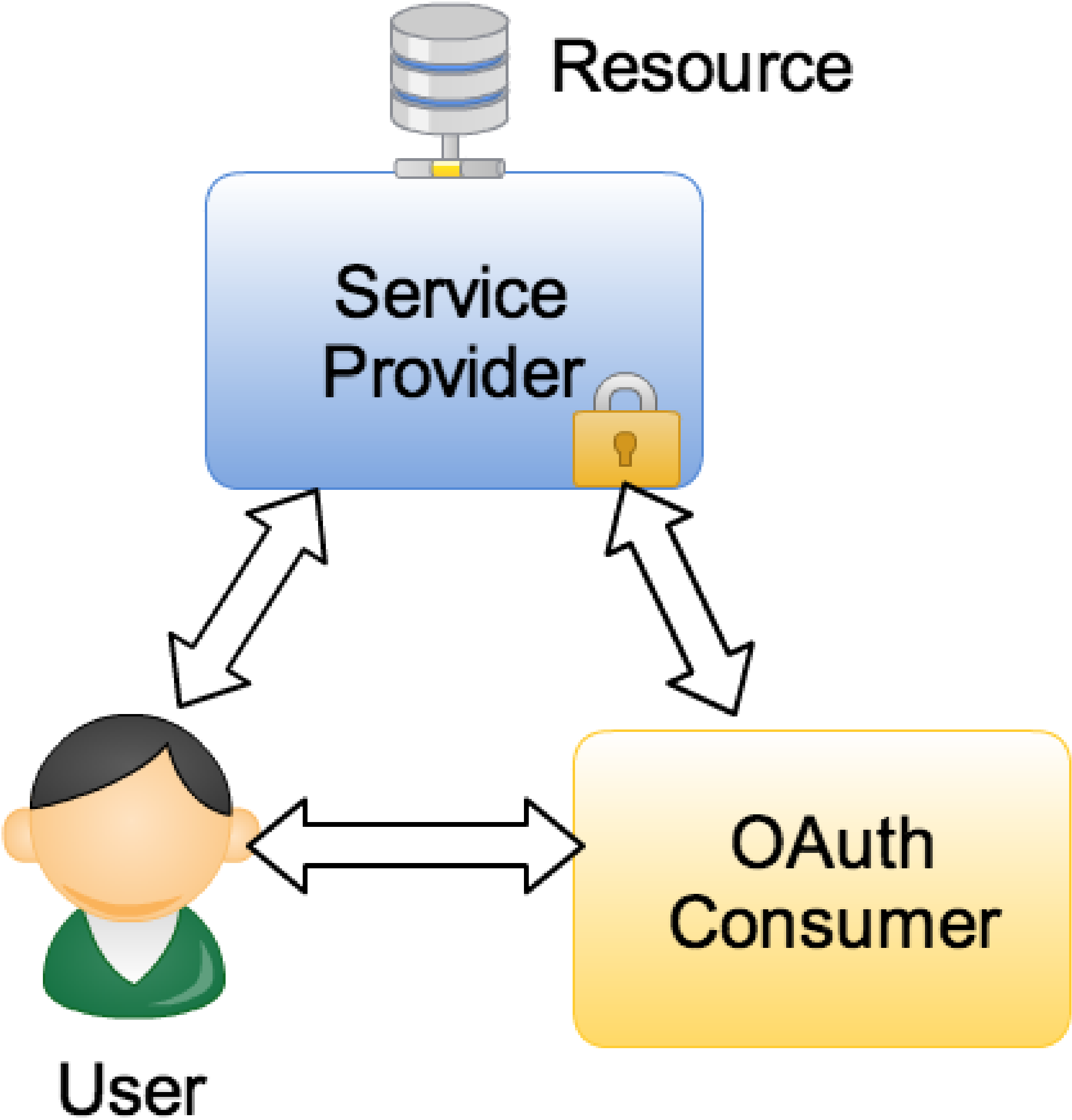


Image: The three different parties in OAuth protocol. The user owns the restricted resource and can grant access to it.

In order to authenticate an user and enabling authorization an OAuth consumer has to be registered at the service provider beforehand. Through registration the service provider obtains a dedicated consumer key & secret pair that is used for authenticating the service whenever user access is requested. This is an significant difference towards the OpenID protocol where identity provider and relying party don't need to know each other and the authentification is done decentral.

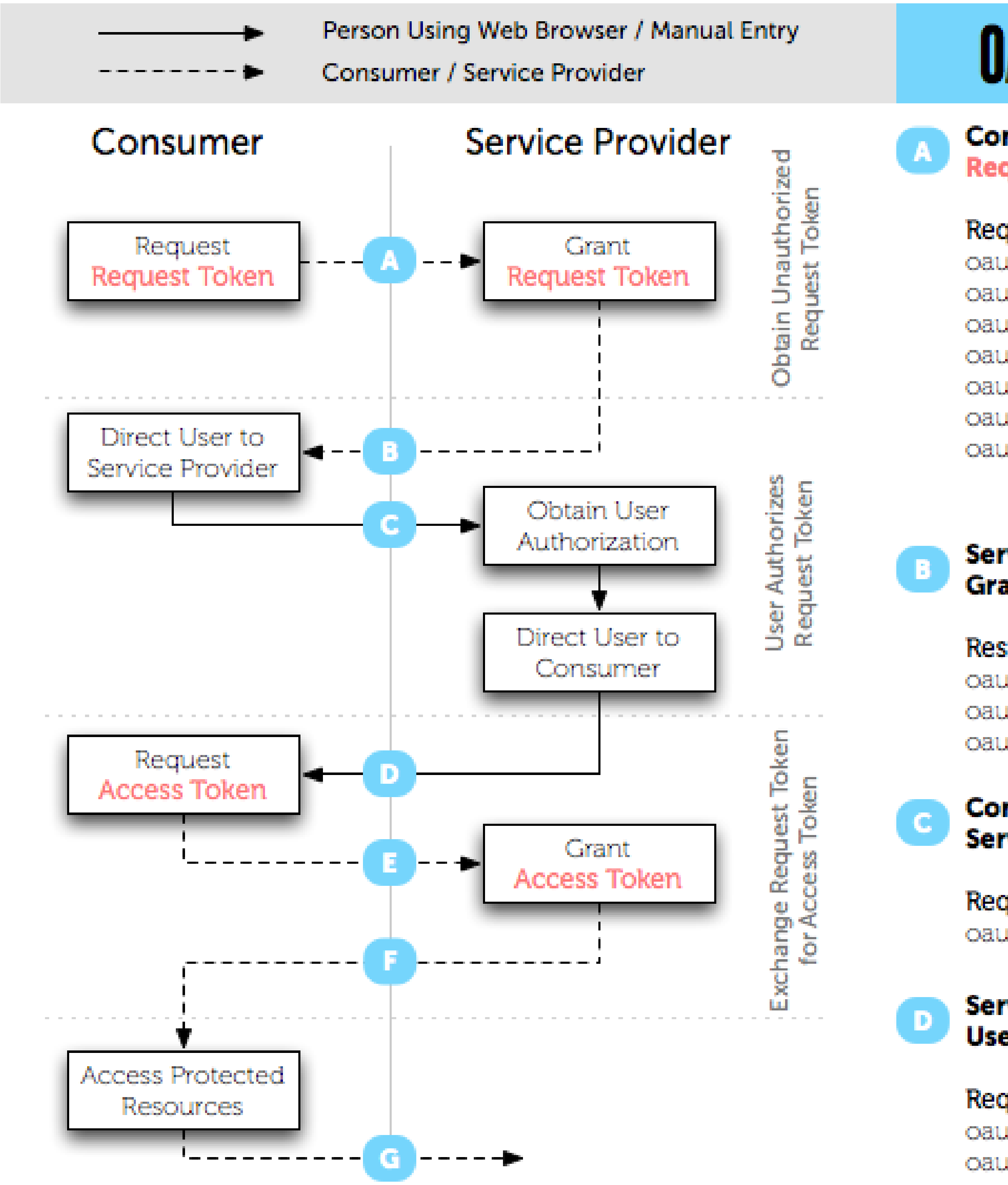
OAuth Dance When a user wants to authenticate a webservice via OAuth the **OAuth authentication flow** (commonly referred to as the OAuth dance) is executed.

The following description is a simplified OAuth description as details important for securing the authorization flow and protect it from replay attacks like the *signature method* and the *nonce* are ommitted. The description is first and foremost meant to provide a general understanding of the OAuth authentication flow as an full description is beyond the scope of this paper.

The authentication flow is started by the user clicking on a special link on the website of the OAuth consumer. The consumer uses his consumer key and secret to request an **unauthorized OAuth request token** and **secret** from the service provider. This OAuth token is used to identify the authentication context for the user.

After obtaining the request token, the user is redirected from the consumer to the service provider. Thereby the request token is appended to the URL. The user is authenticated and can now authorize the consumer. After the user granted access to the restricted resource the service provider directs the user's web agent back to the consumer. A *verifier* is appended to the callback url.

When the user returns to the consumer the consumer uses request token and verifier to request an **access token** and the associated secret from the service provider. The service provider exchanges the request token for the access token thereby granting access to the user's resource(s) as long as the token is valid.


Person Using Web Browser / Manual Entry
Consumer / Service Provider
Consumer
Service Provider
Request
Request Token
A
Grant
Request Token
Obtain Unauthorized Request Token
Direct User to Service Provider
B
C
Obtain User Authorization
Direct User to Consumer
User Authorizes Request Token
Request
Access Token
D
E
Grant
Access Token
F
Exchange Request Token for Access Token
Access Protected Resources
G
A
B
C
D

source: http://dev.twitter.com/pages/auth

1.2 OAuth Privacy EMail Service for HPIIP

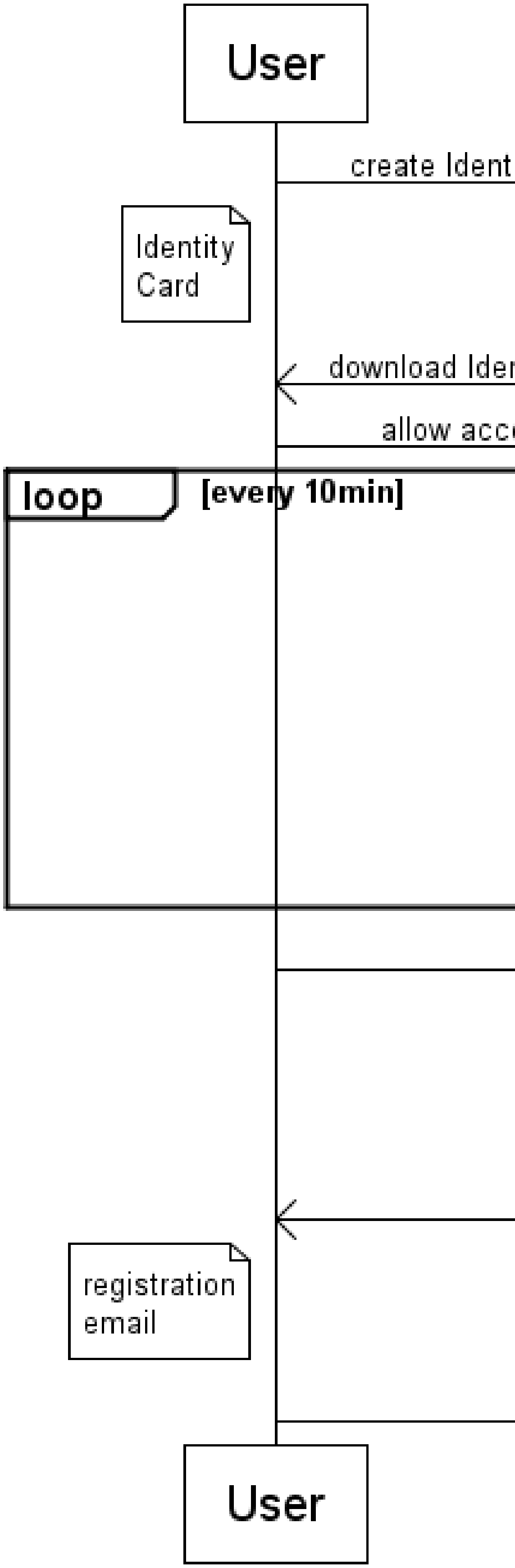
One main feature of the HPIIP is to act as an **identity provider** and issue **IdentityCards**. These Identity Cards can be used by the user to sign up to a **relying party**, allowing the relying party requests associated attribute values from the identity provider. These attributes could be something like the name, home address or the email address of the user.

In order to verify email addresses of new signed up users services often send an email with a verification link. The user has to click this link to verify that he signed up with a valid email address and the user actually owns it.

Often an user only wants to try out a new service but want to ensure his privacy and don't want to provide his real email address in fear of SPAM from the service. Our proposed OAuth example service should be able to change the associated email address value of an identity card when it is authorized by the user.

To grant access to the identity card the user authorize the Privacy EMail Service by clicking on a "Log in to HPI IP" button on the Privacy EMail Service site. The service then acts as a OAuth consumer, redirecting the user's browser to the HPI IP website to login. The user then can grant access of the identity cards to the consumer.

The service changes the value of the email address in an issued identity card every 10 minutes to a temporary valid email address the service has control of. Whenever a user sign up to a relying party with his identity card, the relying party request the current email address attribute from the identity provider and sends an email. The identity provider will return the temporary email adress. Because the OAuth service forwards emails for 20 minutes to the actual email address of the user and ignores emails to the temporary address received thereafter the user will receive only the emails from the relying party that are send in this short time window. All emails received later are considered 'SPAM' and will be ignored.



Below is a sequence diagram of the described behaviour.

Implementing OAuth Privacy Service Our example service has to be a webservice able to receive and send emails. Because we were not in control of and did not want to set up an own SMTP server we choose to use [Google AppEngine].

Google AppEngine is a *Plattform as a Service* infrastructure framework provided by Google. Developers can use Python and Java (in fact every programming language that can be run on the Java VM) to programm web applications for this infrastructure. Like most web frameworks Google AppEngine also implements the *Model-View-Controller (MVC) pattern.* The framework also has APIs to receive and send emails and start Cron jobs.

When an user clicks on the "login with HPIIP" at the service application website he will be redirected to the HPI IP where he can login and grant access to the service. After granting access he then will be redirected back to the application and can complete the setup. The service is now able to access the restricted ressources in behalf of the user.

We can configure a cron job that will call a specific method recurrently every 10 minutes to set up a new temporary email address and use the API of the HPI IP to store it as configured by the user. The same email address is stored by the service and associated with the real email address of the user.

Applications running on Google AppEngine are able to receive and send mails. When the service receives an email it looks for the this address in the temporary email address table. If it exists the email will be redirected to the associated actual email address of the user.

1.3 Implementing OAuth in HPI IP

OAuth defines three request endpoint URLs:

- Request Token URL - we use */oauth/request_token*
- User Authorization URL - we use */oauth/authorize*
- Access Token URL - we use */oauth/access_token*

These endpoints are required to be implemented with the Apache Tapestry5 webframework that is used within the HPI IP.

Request Token URL When the consumer sends an HTTP requests to the Request Token URL */oauth/request_token* there must be the following parameters specified:

- *realm*
- *oauth_consumer_key*
- *oauth_signature_method*
- *oauth_callback*
- *oauth_signature*

The service provider must validate the incoming OAuth message and verify the consumer by checking his credentials against the database. Then the service provider creates a new set of temporary credentials and returns it in a HTTP response body using *"application/x-www-form-urlencoded"* content type.

If the request was valid the response contains the following url encoded parameters:

- *oauth_token*
- *oauth_token_secret*
- *oauth_callback_confirmed*

This oauth token is called the *request token* and it is used to identify the OAuth authorization flow.

User Authorization URL After receiving the *request token* the OAuth consumer redirects the user's web browser to the *authorization url* with the request token appended as parameter **oauth_token**.

The provider has to verify the identiy of the user and then ask to authorize the requested access. Therefor the OAuth provider should display informations about the client based on the request token (like the OAuth client's name, url or logo) for the user to verify.

To prevent repeated authorization attempts the provider has to delete or mark the request token as used. If the user authorize the consumer the user's webbrowser is redirected to the consumer's callback url and *oauth_token* and *oauth_verifier* are appended as parameters.

Access Token URL When the user is redirected to the OAuth consumer the consumer uses the *oauth_verifier* to obtain the permanent access token. To do this the *oauth_verifier* is added to the list of parameters the consumer used to obtain the request token and all parameters are appended to the access token url.

The server must verify the validity of the request. If the request is valid and authorized the permanent token credentials are includes as "application/x-www-form-urlencoded" content type with status code 200 (OK) in the HTTP response. The parameters are

- *oauth_token*
- *oauth_token_secret*

Once the client receives and stores the token credentials it can use it to access protected resorces on behalf of the resource owner. To do so the consumer has to use his credentials together with the access token credentials received.

Implementing OAuth If you have to implement an OAuth consumer you can find many client implementations in every programming language possible. Unfortunately the same does not hold for implementing an OAuth provider. While there are various implementations for scripting languages (notably Ruby, Python and PHP) we have found only one example implementation for Java. This implementation unfortunately isn't a simple to use framework where the developer just provides the persistence layer and the web layer and gets the logic for creating the tokens for free.

1.4 Making API Requests

Whenever the consumer tries to access the user's restricted resource he has to add the OAuth protocol parameters to the request by using the OAuth HTTP "Authorization" header field. The required parameters are:

- *oauth_consumer_key*
- *oauth_token*
- *oauth_signature_method*
- *oauth_timestamp*
- *oauth_nonce*
- *oauth_signature*

The server has to validate the authenticated request by recalculating the request signature, ensuring that the combination of *nonce / timestamp / token* has never used before and verify that the scope and status of the authorization as represented by the OAuth request token is valid.

Granularity of Rights Management While most existing OAuth providers (e.g. Twitter) manage rights only at the granularity of of the access token allowing or denying read/write access to every of the user's resources, it is often feasible to manage rights with finer granularity. In fact this is what we need to do in the HPI IP in order to allow users to grant access to some relevant attributes of their identities and deny access to others.

By this means that not only the API consumers use is becoming more complex, the complexity for the user interface to manage access rights for consumers transparently increases too. If this problem isn't solved in the user interface it could lead to conflicts over privacy issues like it is the case with Facebook's third party apps.

In order to have maximum fine granularity of rights management we decided to manage rights to every of attributes of each digital identity seperatly.

Accessing the User's Restricted Resources In today's internet world there are many different technologies for providing API access but RESTful Web Services prevail. We decided to build the neccessary API in a restful way using URLs to name resources, HTTP verbs for methods and HTTP response codes to signal results. [REST] This way we have to decide which resources should be available and how they are represented.

The OAuth access token is essentially the right to access some of the associated user's restricted resources. The OAuth client has to query the API to find out which specific resources it can access in behalf of the user. Therefor we need one URL endpoint that is the same for every user. By querying it with the access token the consumer specifies the user and gets some more information about which resources are accessable with the provided token.

- the resource of the user who granted the access token should be *HPIIP/api/user*

The representation of this resource in JSON should be something like this, showing only the values that are available to the requesting consumer:

```
{
    "user" : {
        "username" : "richard.metzler",
        "identities" : [
            {
                "main identity" : [
                    {
                        "id" : "123456",
                        "attribute" : "email",
                        "value" : "richard@...",
                        "resource" : "HPIIP/api/attribute/123456",
                        "writable" : true
                    }, ...
                ]

            }, ...
        ]
    }
}
```

As you can see, the 3rd party service is able to find out the resource for reading and updating the email resource. Reading is done with an HTTP GET request, while updating is by sending an HTTP POST request to the *HPIIP/api/attribute/{id}* resource. The resource is again represented in JSON format:

```
{
    "id" : "123456",
    "attribute" : "email",
    "value" : "richard@...",
    "resource" : "HPIIP/api/attribute/123456",
    "writable" : true
}
```

If the client updates the value the attribute only the attribute *value* needs to be in representation.

Whenever an OAuth consumer wants to access a resouce the OAuth provider has to verify if the requested operation is granted to provided access token. It has to be denied otherwise using the HTTP response code **401 ("Unauthorized")**.

Implementation The proposed implementation of the described API is to use *Jersey*, Sun's implementation of the JAX-RS specification. JAX-RS describes a Java API to build RESTful webservices by using Java5 style annotations. These webservices have to run in a servlet container like Apache Tomcat or Jetty.

Because the existing HPI IP uses the Tapestry5 webframework we had to decide if the API and the existing application should be deployed next to each other and use their own database bindings or deploy it as one application in the same directory. Because we wanted to reuse the existing database bindings we decided to use the Tapestry-Jersey integration provided by Blue Tang Studio.[1][2]

References