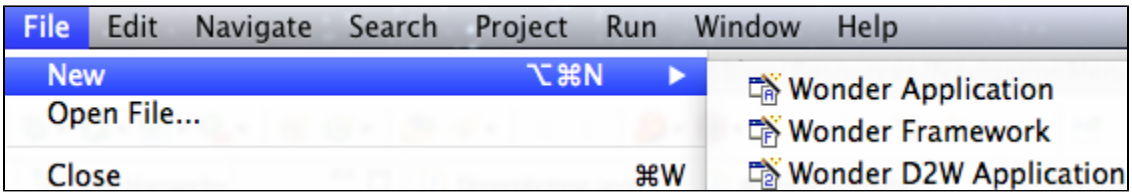
Your First D2W Project

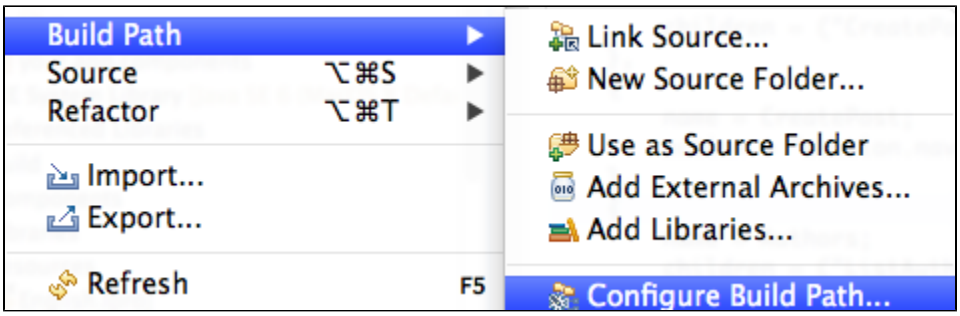
Work in progress

Now that our data model is in a framework and that we have a basic REST application working, we are going to build a DirectToWeb application to manage the blog system. But first, what is DirectToWeb (D2W)? D2W is a rules-based application model where you can change the behavior of the application by rules. D2W is perfect for "admin" apps, for applications that share a common model or for "**CRUD**" (**C**reate **R**ead **U**pdate **D**estroy) applications.

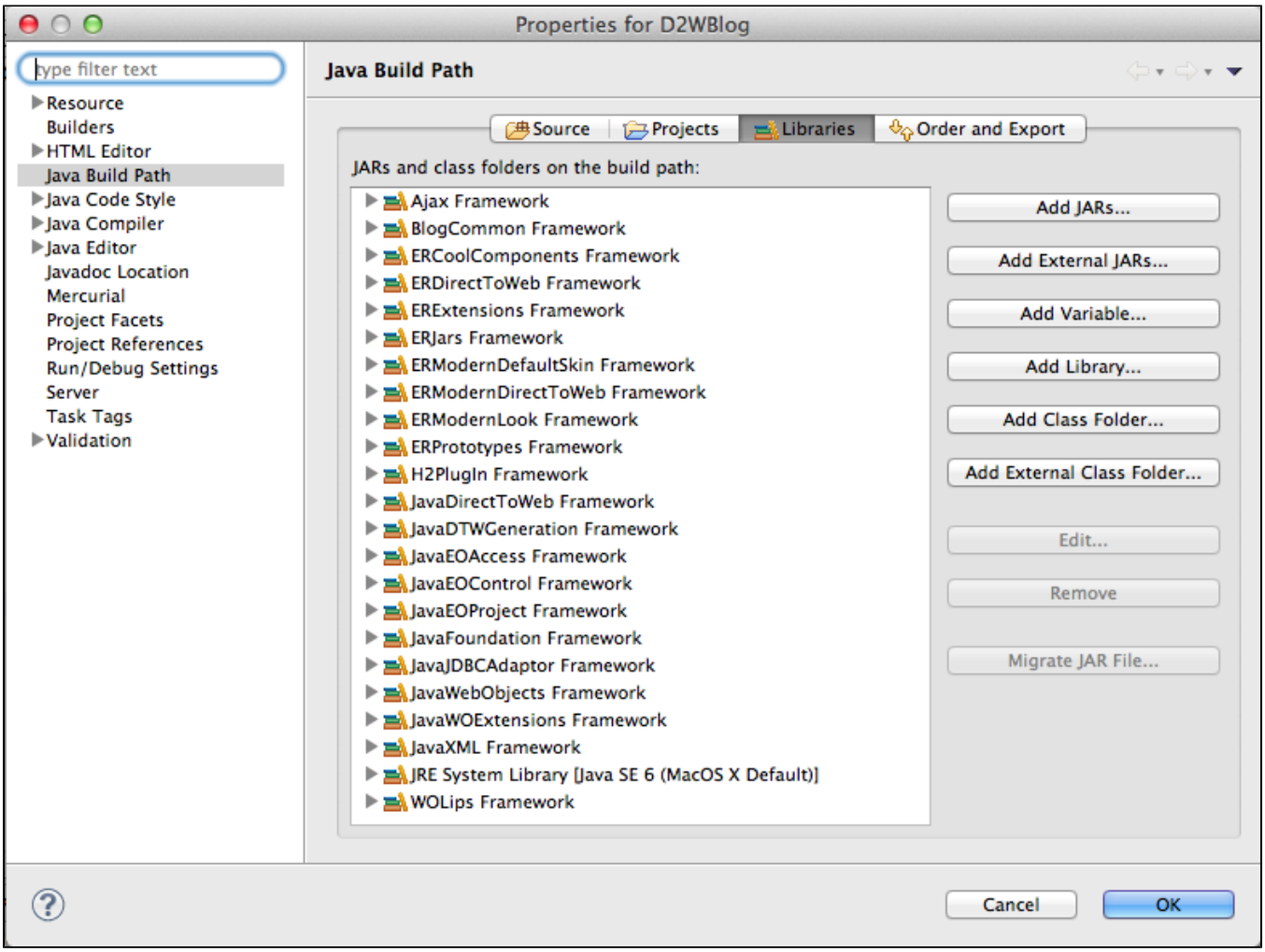
So like I said, we are going to build a D2W app that will allow us to update and create blog entries and authors. Let's start by creating a *Wonder D2W Application* in Eclipse:



Name it **D2WBlog**. The next step is to link the **BlogCommon** framework to the D2W app. To do so, right-click on **D2WBlog** and select **Build Path** -> **Configure Build Path**.



In the **Libraries** tab, click on **Add Library**. Select **WebObjects Frameworks** and click **Next**. Check **BlogCommon** and **H2PlugIn** from the list and click **Finish**. The **Libraries** tab should look like this:



It's time to run the app. Right-click on **Application.java** and select **Run As** -> **WO Application**. By default, a D2W app will always display a login form and the login method is connected to a DirectAction that should handle the authentification. The default implementation of that DirectAction (the loginAction method in the DirectAction class) don't do anything special:

```
public WOActionResults loginAction() {
    String username = request().stringFormValueForKey("username");
    String password = request().stringFormValueForKey("password");
    return D2W.factory().defaultPage(session());
  }
```

Just click the **Login** button and you will get into the application. Navigate in the application, but you will see that it don't do anything special and you don't see data. You can terminate the app (by clicking the red stop button in Eclipse's console).

The first thing we will do is to make authentification to work. We don't have a password in our data model so authentification will only be done on the email address (yes, we know it's not very secure). The first thing we need to do is to store the logged author into the session so that we can access it from all pages. To do so, open the **Session** class and add the following code:

```
import er.extensions.foundation.ERXThreadStorage;

  private Author _author;

  public Author author() {
    return _author;
  }

  public void setAuthor(Author author) {
    _author = author;
    ERXThreadStorage.takeValueForKey(author(), "author");
  }
```

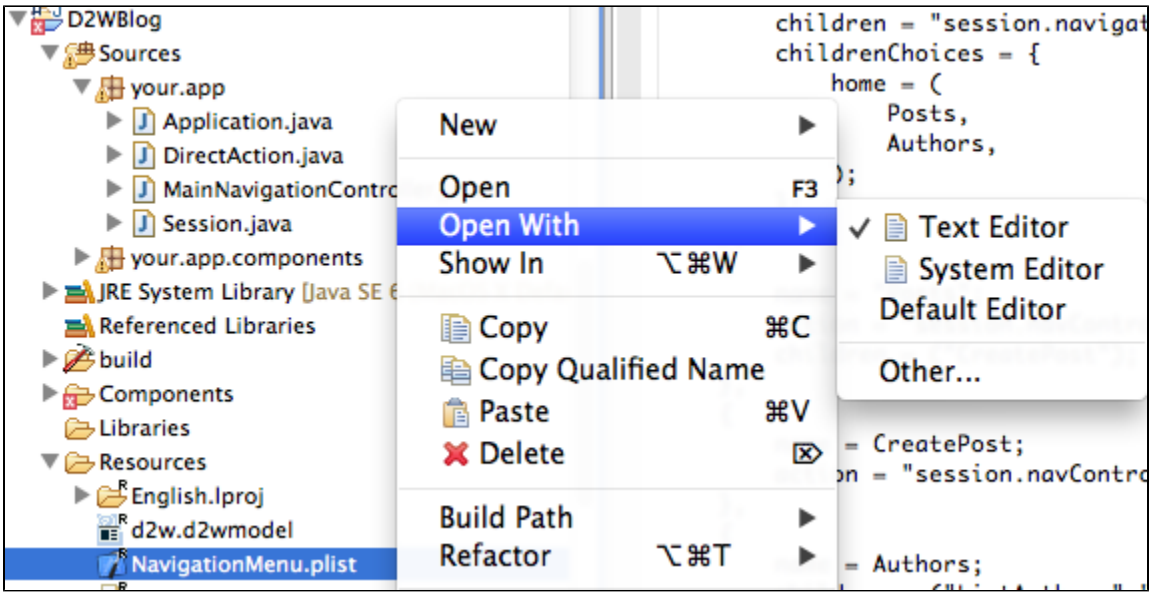
Open the **DirectAction** class and rewrite the **loginAction** method like this:

```
import er.extensions.eof.ERXEC;
import er.extensions.foundation.ERXStringUtilities;
import your.app.model.Author;

  public WOActionResults loginAction() {
    WOComponent nextPage = null;
    String email = request().stringFormValueForKey("username");
    String errorMessage = null;

    try {
      Author user = Author.fetchAuthor(ERXEC.newEditingContext(), Author.EMAIL.eq(email));
      ((Session) session()).setAuthor(user);
      nextPage = ((Session) session()).navController().homeAction();
    }
    catch (NoSuchElementException e) {
      errorMessage = "No user found for that combination of username and password.";
    }
    catch (Exception e) {
      // TODO: handle exception
    }
    if (!ERXStringUtilities.stringIsNullOrEmpty(errorMessage)) {
      nextPage = pageWithName(Main.class);
      nextPage.takeValueForKey(errorMessage, "errorMessage");
      nextPage.takeValueForKey(email, "username");
    }
    return nextPage;
  }
```

The **loginAction** logic is quite simple: we check the value from the form, we check in the data source if the email address exists, if that's the case we store it in the session and we redirect the user to the default (*navController().homeAction()*) page, if now the user stays on the Main page.

We are done with the login logic, so we can move to the next step: working on the navigation. The navigation is quite simple: we want two tabs, *Posts and Authors*, when one of the two tabs is selected, we display the list of objects for the selected tabs, and we have two links: one to create a new blog entry or a new author, and the other to query (search) for matching objects.

The navigation structure is done in a "plist" file. If you are a Mac OS X guy, you probably know what is a plist, but if not, a plist is like a JSON structure to hold key/values. The plist file is located in the **Resources** folder, the file's name is **NavigationMenu.plist**. By default, Eclipse will open the file in Xcode, which might not be what we want. To open it in a text editor in Eclipse, right-click on **NavigationMenu.plist** and select **Open With** -> **Text Editor**.



Edit the file with the content is like this:

```
(
        {
                name = Root;
                directActionClass = DirectAction;
                directActionName = default;
                children = "session.navigationRootChoice";
                childrenChoices = {
                        home = (
                                Posts,
                                Authors,
                        );
                };
        },
        {
                name = "Posts";
                action = "session.navController.listPostsAction";
                children = ("CreatePost","SearchPosts");
        },
        {
                name = CreatePost;
                action = "session.navController.createPostAction";
        },
        {
                name = SearchPosts;
                action = "session.navController.searchPostsAction";
        },
        {
                name = Authors;
                action = "session.navController.listAuthorsAction";
                children = ("CreateAuthor","SearchAuthors");
        },
        {
                name = CreateAuthor;
                action = "session.navController.createAuthorAction";
        },
        {
                name = SearchAuthors;
                action = "session.navController.searchAuthorsAction";
        }
)
```

Add the following method in the **Session** class for the navigation root.

```
public String navigationRootChoice() {
        return "home";
}
```

The first array in the plist defines what the top level navigation is going to be, and this is where we define the two tabs (the *childrenChoice* dictionary). After that, we define the other parts of the navigation. You see references to *action = session.navController*. This is the action (method) that will be called for the specified navigation element, so let's create those methods in the **MainNavigationController** class.

The first method we will implement in **MainNavigationController** is a generic method to list objects for a specific entity.

```
public WOComponent listPageForEntityName(String entityName) {
    ListPageInterface listPage = D2W.factory().listPageForEntityNamed(entityName, session());
    EODataSource dataSource = new EODatabaseDataSource(session().defaultEditingContext(), entityName);
    listPage.setDataSource(dataSource);
    return (WOComponent) listPage;
  }
```

The controller already have methods to query objects (*queryPageForEntityName*) and create new ones (*newObjectForEntityName*), so the next step is to create the methods for our two entities.

```
public WOComponent listPostsAction() {
    return listPageForEntityName(BlogEntry.ENTITY_NAME);
  }

  public WOComponent listAuthorsAction() {
    return listPageForEntityName(Author.ENTITY_NAME);
  }

  public WOComponent createPostAction() {
    return newObjectForEntityName(BlogEntry.ENTITY_NAME);
  }

  public WOComponent createAuthorAction() {
    return newObjectForEntityName(Author.ENTITY_NAME);
  }

  public WOComponent searchAuthorsAction() {
    return queryPageForEntityName(Author.ENTITY_NAME);
  }

  public WOComponent searchPostsAction() {
    return queryPageForEntityName(BlogEntry.ENTITY_NAME);
  }
```

We also need to change the *homeAction* method so that when an user log ins, he see the list of blog entries.

```
public WOComponent homeAction() {
    return listPageForEntityName(BlogEntry.ENTITY_NAME);
  }
```

Save the file and run the app.

After login, you will see the blog entries and if you click the **Authors** tab, you see the list of authors. Each item in the list has 3 actions by default: **Inspect** (view the object), **Edit** (modify the object) and the red X button to delete the object.

Click **Edit** on an author, and you will see that it displays not only the author's details but also blog entries created for that user. You can even create a new blog entry directly from the author.

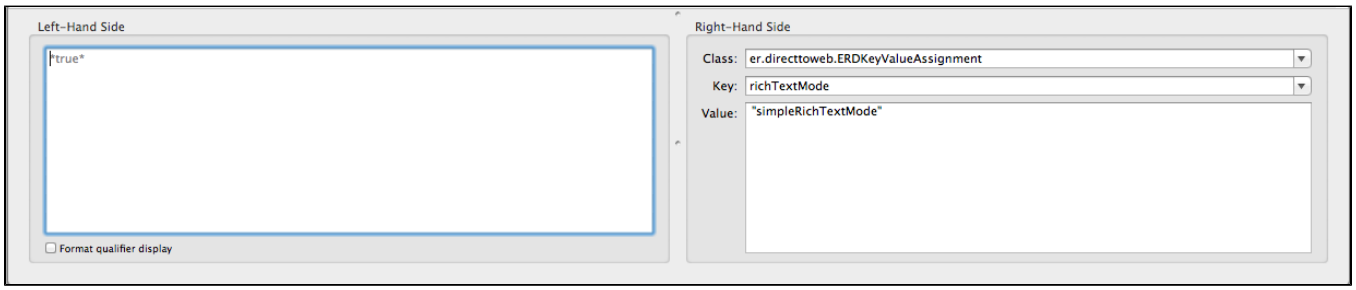
Now, edit a blog entry. If you click on the field next to the **Creation Date** or **Last Modified**, a calendar widget appears because those two fields are marked as dates in the data model, that's EOF and D2W magic at work.

Viewing or editing a blog entry can be improved. The three things we want to customize:

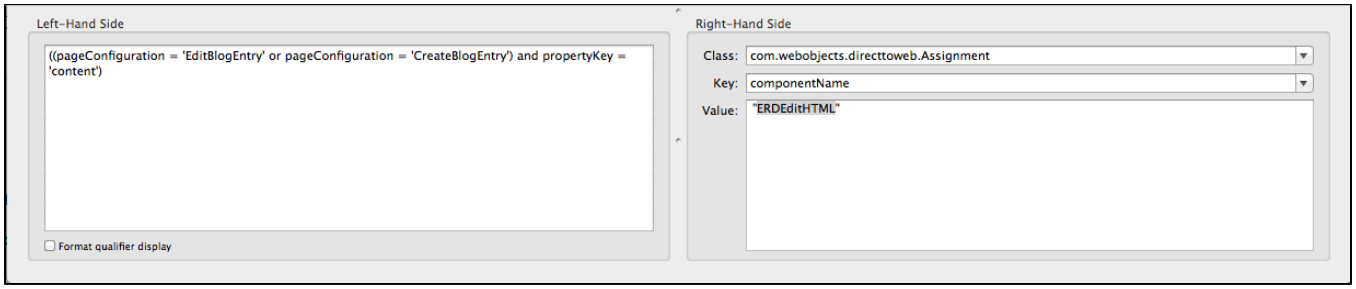
- to remove the email address of the author when viewing the blog entry
- to make the **Content** field to be a text area instead of a input field, and even better: attach TinyMCE to the text area
- modifying the order of attributes when viewing or editing a blog entry so that the title field is the first field, followed by content, the two dates and the author

Those two customizations can be done by adding D2W rules. The D2W rules are in two files, located in the **Resources** folder, **d2w.d2wmodel** and **user.d2wmodel**. To edit the files, make sure you installed RulesModeler, a Mac application that manages D2W rule files. If RuleModeler is present, you can simply double-click on **d2w.d2wmodel** and the file will open in RulesModeler.

The first rule we need to add is a rule to specify that the *richTextMode* will be _ simpleRichTextMode_, which is a built-in definition that will put TinyMCE in a simple configuration. Create the **New** button in RulesModeler, and the rule must looks like this:



The next rule will specify that we want to use the EREditHTML component for the *content* attribute. Again, click **New** in RulesModeler, and add this rule:



Last rule: when we want to inspect or edit the blog entry, we want to change how the attributes are displayed, so add this new rule:

We are done. Save the file in RulesModeler, and run the application again. Try editing a blog entry and you will notice that we know have a rich editor for the content and that the ordering of the fields is different, without having to change the HTML or the Java code!

Congratulations, you are done with the D2W tutorial! The next tutorial is about creating a stateful application.