Eclipse RCP MailApp 4.0

Tom Schindl <tom.schindl@bestsolution.at>

Inhaltsverzeichnis

It's one of the most used RCP-Applications to teach people the concept of the 3.x platform is the mail demo generated by the PDE-Wizard. In this tutorial we are going to create as a first step a similar application using technologies from e4.

Setup the IDE

Probably the most natural way to develop an e4-RCP application is to download the Eclipse 4.0 SDK which uses the e4 runtime platform to provide you and Java and OSGi-Tooling-IDE.

We should mention at this point that you are NOT forced to use Eclipse 4.0 to write e4-RCP applications and all the introduced tooling is available to you as well in the Helios Release through the e4-Update-Site of the Eclipse-IDE.

Though we appreciate if you use Eclipse 4.0 as you IDE we'd like to mention that it is not primarily targeted for daily work but marked as an „Early Adopter Release" giving plugin developers the possibility to test if their 3.x Bundles run in a 4.0 environment.

Download & Install Eclipse 4.0 SDK

You should be able to download Eclipse 4.0 from http://www.eclipse.org/helios/eclipse-sdk-4.0/. Featurewise what you get with this download is comparable to Eclipse Classic 3.6 which includes JDT, PDE, CVS, ...

After having download the Eclipse version for your platform you'll have to unzip it and launch the platform executeable and you should see an 4.0 SDK similar to this one.

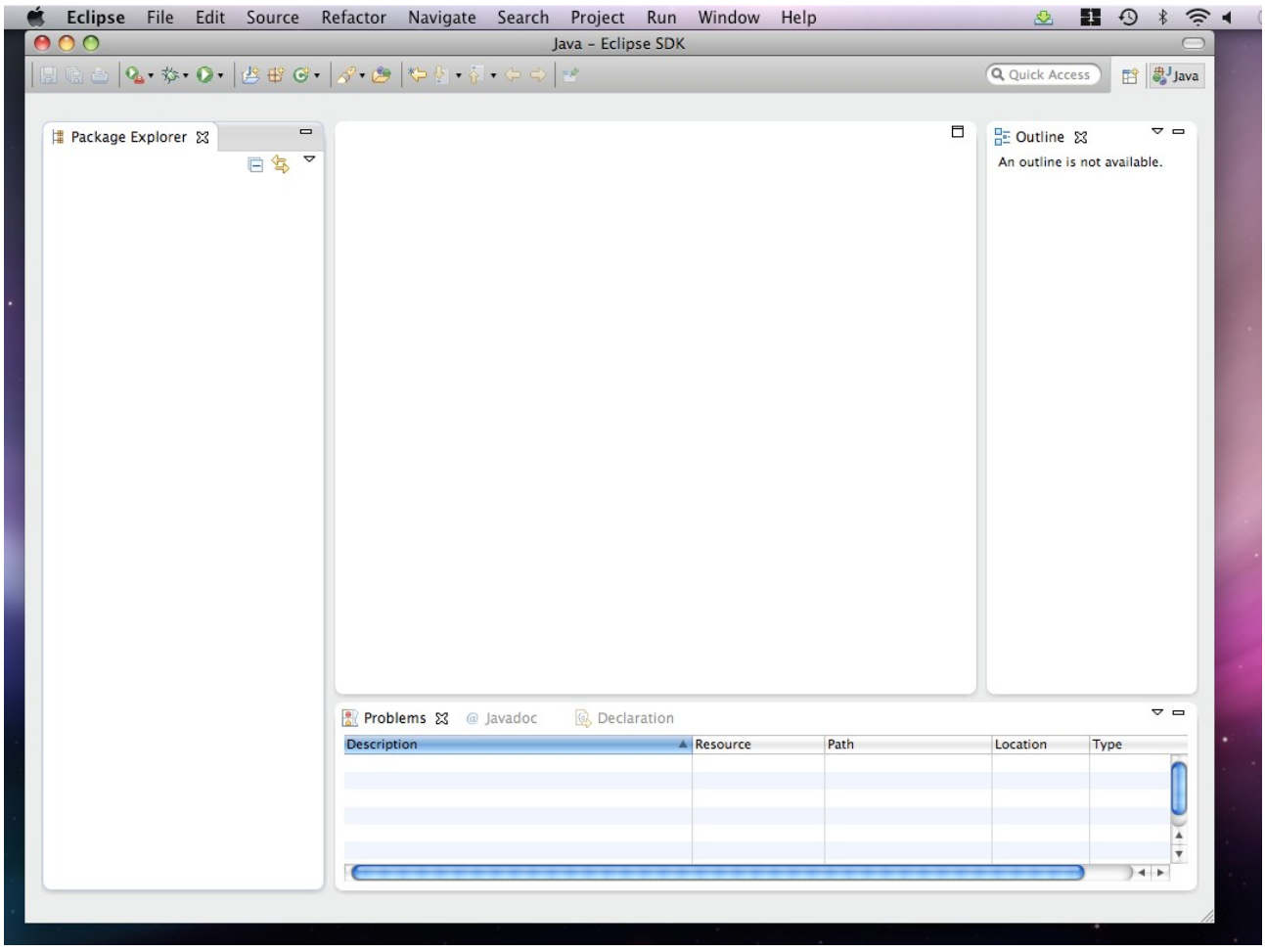


Abbildung 1: 4.0 SDK

Install e4 (Model)Tooling

This is not needed if you are familiar with XMI and want to edit files like this

using a standard text-editor but most people prefer to use tooling which helps them writing applications.

Because the e4 Tooling has not yet graduated to 4.0 SDK you need to install it using Help > Install New Software … .

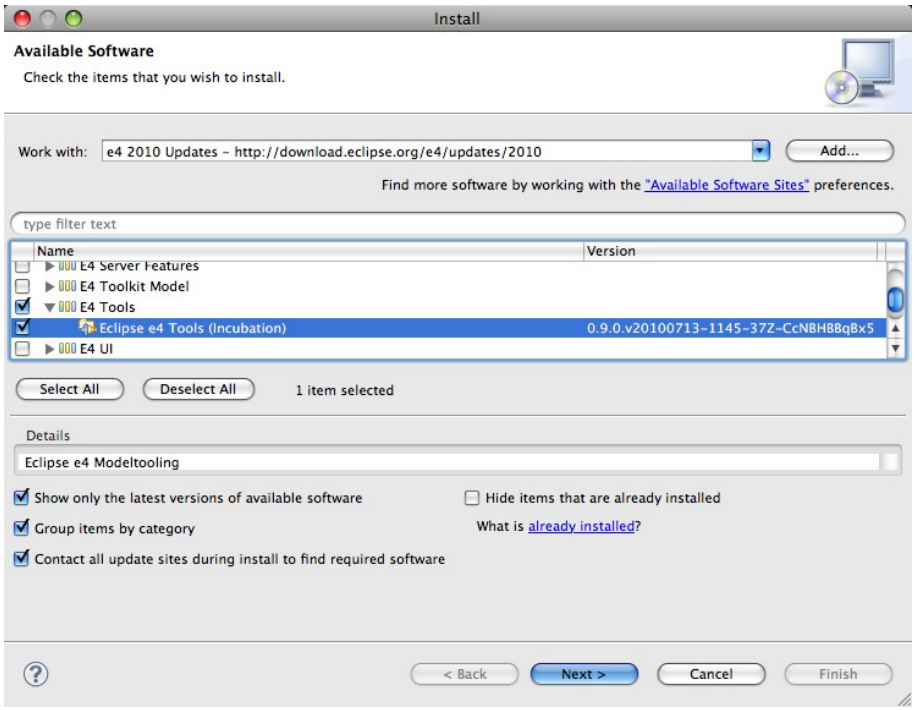


Abbildung 2: Install e4 Tooling

Setup up project

There is a wizard to create a complete e4-RCP-Project but we are not using this wizard but setup the project by hand so that we understand step by step what's done.

Create an OSGi-Project

We are using File > New > Project … and select Plug-in Development > Plug-in Project and enter the following data into the wizard pages:

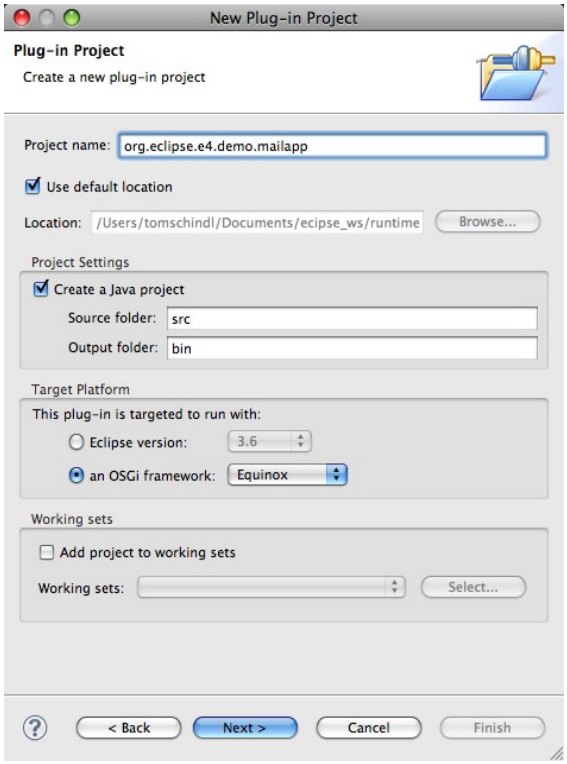


Abbildung 3: New OSGi-Project 1

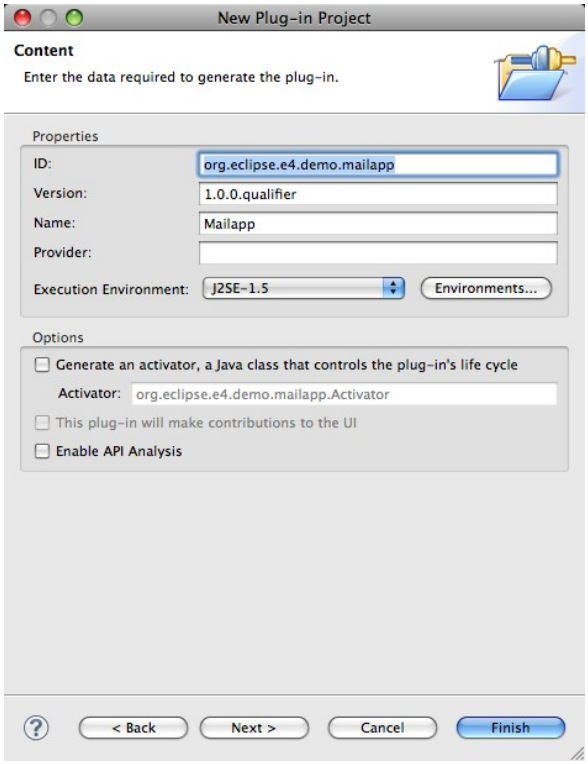


Abbildung 4: New OSGi-Project 2

Add a product definition

Create something we can launch easily what one does when using Equinox is to create an application and product definition using the extension points provided by „org.eclipse.equinox.app".

To write an e4-Application we don't have to define our own application but reuse an application already defined by the e4 runtime named „org.eclipse.e4.ui.workbench.swt.E4Application".

Let's do things step by step:

a) Open the MANIFEST.MF

Add a dependency on „org.eclipse.equinox.app"

b) Open the plugin.xml

Add a product definition like this

```
<extension
       id="product"
       point="org.eclipse.core.runtime.products">
  <product
       application="org.eclipse.e4.ui.workbench.swt.E4Application"
       name="Mail App">
  </product>
</extension>
```

What's missing now to launch our application for the first time are 2 things we

need to do. An e4-Application which uses the predefined E4Application has to have a minimal workbench model (we'll learn about this in the next sections), and a .product to define a launchable application.

Add a minimal e4-ApplicationModel

In contrast to 3.x application where you used a mixture of Java and Extension Points to setup up an application e4 applications follow another route. The complete application is defined and made up from one single model.

You'll learn in later sections of the tutorial how this application model can be made up dynamically but for getting something up and running we'll create a minimal application model using „File > New > Other" and select „e4 > Model > New Application Model".

Fill in the following informations:

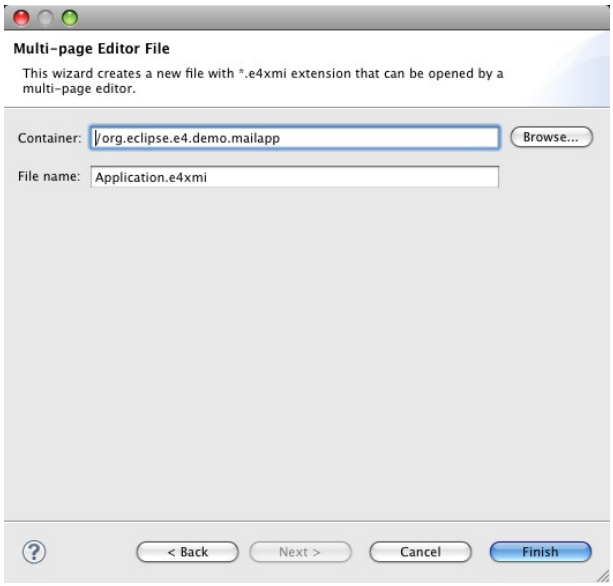


Abbildung 5: New Application Model

Technically this would be enough to launch an application but an UI-Application without at least one window is senseless.

After having created the Application.e4xmi the „e4 Workbench Model"-Editor should have opened itself automatically.

Select the „Windows" entry on the left, select „TrimmedWindow" on the right and press the button next to the drop down. The result should be an editor looking like this:

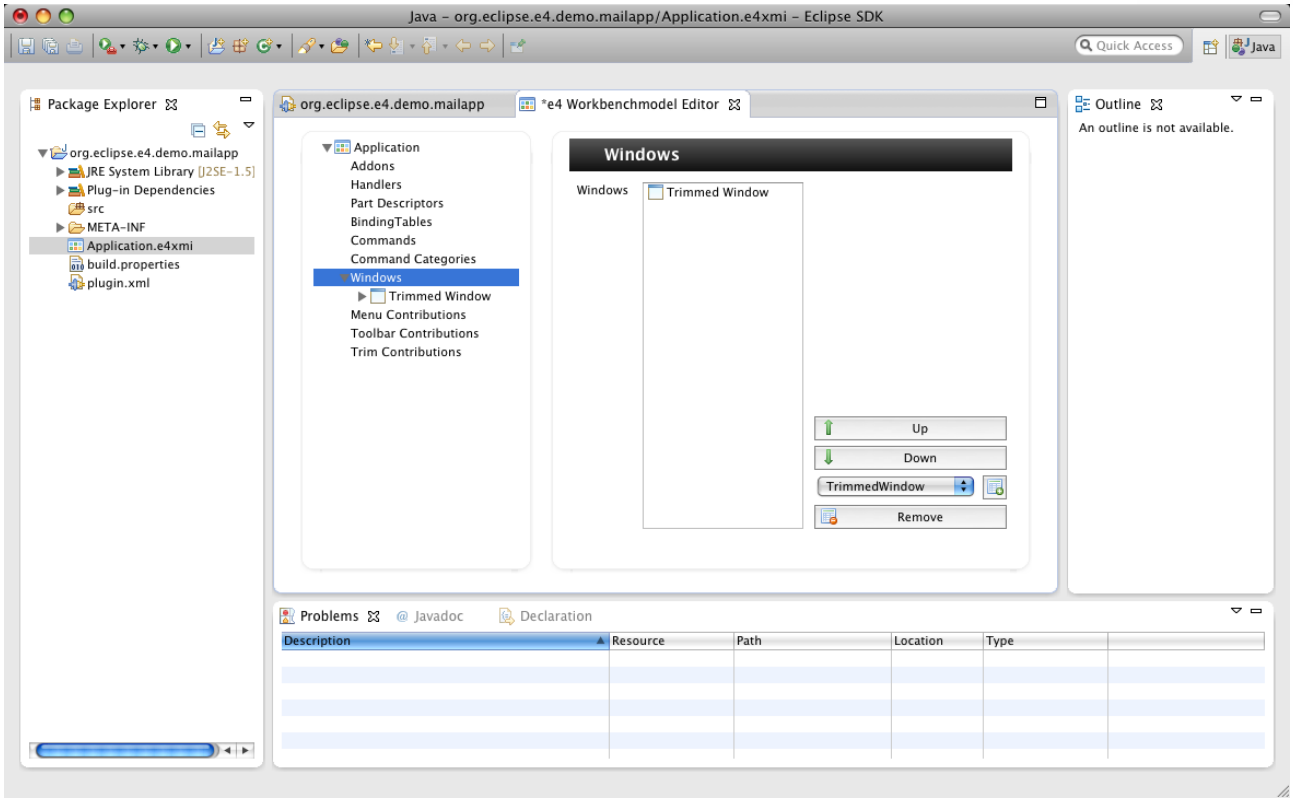


Abbildung 6: Add window to model

Afterwards select the „Trimmed Window" entry in the tree and set the height and width values to 640 and 480 and the Label-Property to „MailDemo 4.0".

Create a MailDemo-4.0.product

A product file allows us to define a product we'll export later on to provision on our clients desktops. If you should familiar with the process of creating such a .product but here's a step by step instruction because we need to add some extra stuff PDE is not able resolve itsown.

1. New > File > Other …
2. Plug-in Development > Product Configuration
3. In dialog enter:
   - Filename: MailDemo-4.0
   - Use an existing product: org.eclipse.e4.demo.mailapp.product
4. Add the following additional bundles
   - org.eclipse.equinox.ds
   - org.eclipse.equinox.event
   - org.eclipse.e4.ui.workbench.renderers.swt

5. Press „Add Required Plug-ins“

Before we can launch we need to add some more information to our product-extension point to make it look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension
         id="product"
         point="org.eclipse.core.runtime.products">
      <product
            application="org.eclipse.e4.ui.workbench.swt.E4Application"
            name="Mail App">
         <property
               name="appName"
               value="Mail App">
         </property>
         <property
               name="applicationXMI"
               value="org.eclipse.e4.demo.mailapp/Application.e4xmi">
         </property>
      </product>
   </extension>

</plugin>
```

The important information we need to provide to the E4Application is which initial model it should use to make up the application.

Now we are ready to launch our minimal e4 application the first time and it will show us something like this:

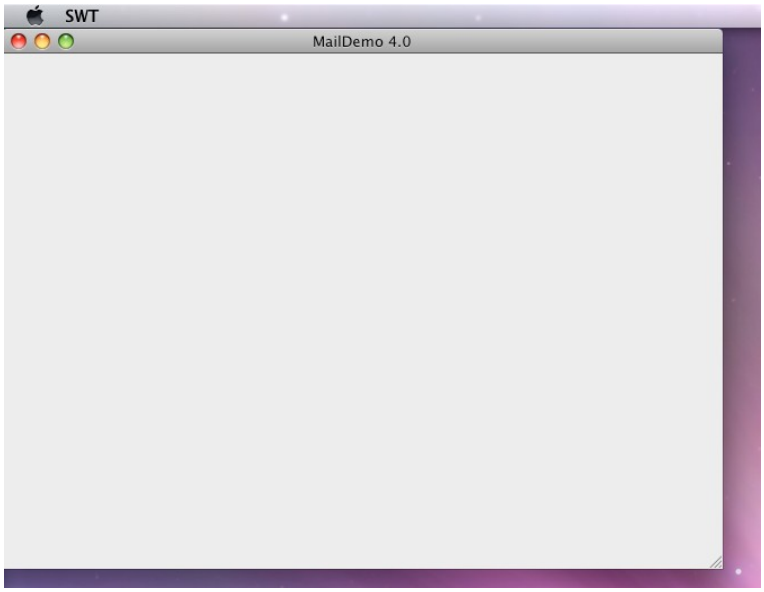


Abbildung 7: First Application

TODO: Provide source code after Chapter 2 as a zip

Create the MailServices

To let our application really do meaningful stuff and present you all the cool new features e4 provides you when writing OSGi-base UI-Applications we are going to add some OSGi-Service stuff.

The e4-runtime itself is designed from day one with OSGi in mind and to follow good OSGi-practices we create 2 new OSGi-projects:

- org.eclipse.e4.demo.mailapp.mailservice
- org.eclipse.e4.demo.mailapp.service.mock

I'm not going into detail here how this is implemented but you should simply download the ready bundles and import them in your workspace.

TODO: Location where user can download 2 bundles as a zip

The important APIs for now are:

- `IMailSessionFactory#openSession()`: Which allows you to open a mail session
- `IMailSession#getAccounts()`: To retrieve mail accounts
- `IMailSession#getMails()`: To fetch mails from a folder

If you are not familiar with Declarative OSGi-Services there's a vast number of tutorials and books out describing them in great detail.

After having imported the bundles your workspace should look like this:

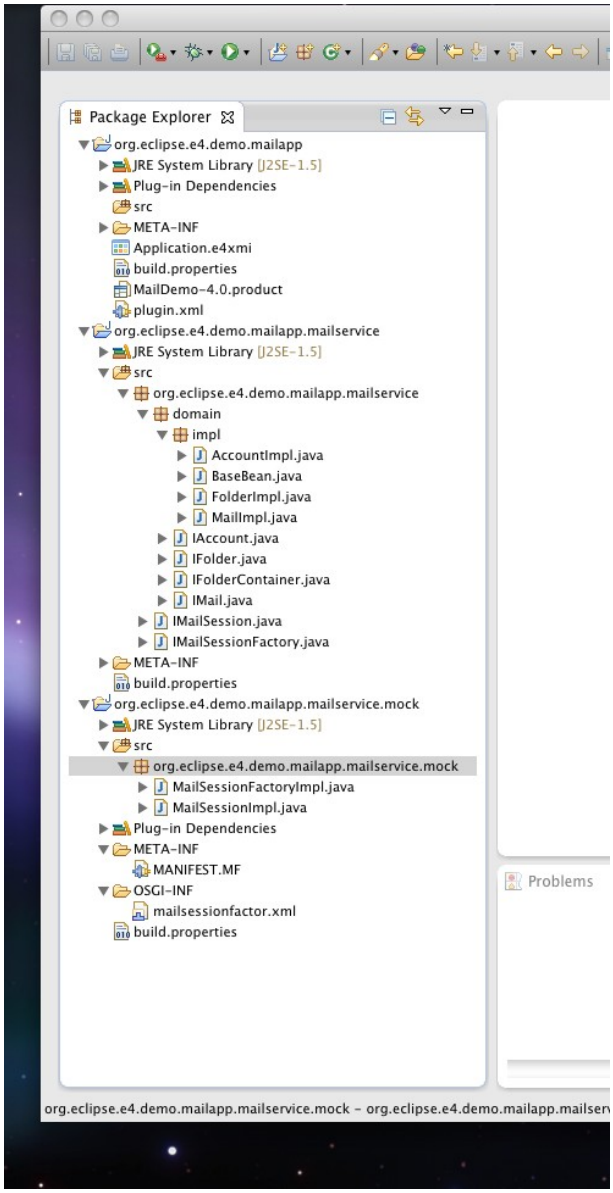


Abbildung 8: Workspace after ServiceBundles import

To finish this task we need to add the 2 new bundles to our .product-File and recreate our launch configuration so that those new bundles are picked up.

Create the UI

Next thing we need to do is to make write our UI-Code. In 3.x we would have derived our UI-Parts from ViewPart or EditorPart but this is not needed anymore for e4 where everything is a POJO.

Before we can start writing our UI code we need to add the following dependencies to our MANIFEST.MF in „org.eclipse.e4.demo.mailapp“:

- org.eclipse.swt

- org.eclipse.jface
- org.eclipse.jface.databinding
- org.eclipse.core.databinding
- org.eclipse.core.databinding.observable
- org.eclipse.core.databinding.property
- org.eclipse.core.databinding.beans (this one you also have to add .product-File – don't forget to update your launch-config!)
- org.eclipse.e4.demo.mailapp.mailservice

Implement the AccountView UI

Next we create a new Java-Class named org.eclipse.e4.demo.mailapp.AccountView and add the following lines of Java-Code into it.

```
package org.eclipse.e4.demo.mailapp;

import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.IObservable;
import org.eclipse.core.databinding.observable.list.IObservableList;
import org.eclipse.core.databinding.observable.masterdetail.IObservableFactory;
import org.eclipse.core.databinding.property.list.IListProperty;
import org.eclipse.e4.demo.mailapp.mailservice.IMailSession;
import org.eclipse.e4.demo.mailapp.mailservice.IMailSessionFactory;
import org.eclipse.e4.demo.mailapp.mailservice.domain.IAccount;
import org.eclipse.e4.demo.mailapp.mailservice.domain.IFolder;
import org.eclipse.e4.demo.mailapp.mailservice.domain.IFolderContainer;
import org.eclipse.jface.databinding.viewers.ObservableListTreeContentProvider;
import org.eclipse.jface.databinding.viewers.TreeStructureAdvisor;
import org.eclipse.jface.viewers.ColumnLabelProvider;
import org.eclipse.jface.viewers.TreeViewer;
import org.eclipse.swt.widgets.Composite;

public class AccountView {
  private IMailSessionFactory mailSessionFactory;
  private IMailSession mailSession;
  private TreeViewer viewer;
  private String username = "john";
  private String password = "doe";
  private String host = "tomsondev.bestsolution.com";

  public AccountView(Composite parent, IMailSessionFactory mailSessionFactory) {
    this.mailSessionFactory = mailSessionFactory;
    viewer = new TreeViewer(parent);
    viewer.setLabelProvider(new ColumnLabelProvider() {
      @Override
      public String getText(Object element) {
        if( element instanceof IAccount ) {
          return ((IAccount) element).getName();
        } else if( element instanceof IFolder ) {
          return ((IFolder)element).getName();
        }
        return super.getText(element);
      }
```

```
    });

    IObservableFactory factory = new IObservableFactory() {
      private IListProperty prop = BeanProperties.list("folders");

      public IObservable createObservable(Object target) {
        if( target instanceof IObservableList ) {
          return (IObservable) target;
        } else if( target instanceof IFolderContainer ) {
          return prop.observe(target);
        }
        return null;
      }
    };

    TreeStructureAdvisor advisor = new TreeStructureAdvisor() {};

    viewer.setContentProvider(new ObservableListTreeContentProvider(factory, advisor));
  }

  public void setUsername(String username) {
    this.username = username;
  }

  public void setPassword(String password) {
    this.password = password;
  }

  public void setHost(String host) {
    this.host = host;
  }

  public void init() {
    if( username != null && password != null && host != null ) {
      mailSession = mailSessionFactory.openSession(host, username, password);
      viewer.setInput(mailSession.getAccounts());
    }
  }
}
```

Now how can we test this UI? We could add it directly to our RCP-Application but when looking closer to it we see that there's no need to bring up the complete framework to see what our part is doneing.

There's not even a dependency on an OSGi-Environment so the class above should be runnable as a standard Java-Application.

Create a TestProject

We create a Test project we can use to launch our UI-Codeparts who have now no real dependency on the Application-Framework nor OSGi itself.

Although we are writing a standard Java application lets create a PDE-enabled project named „org.eclipse.e4.demo.mailapp.test“ so that we don't have to manage the classpaths our own.

Add the following dependencies to the MANIFEST.MF:

- org.eclipse.swt
- org.eclipse.jface.databinding
- org.eclipse.core.databinding
- org.eclipse.e4.demo.mailapp
- org.eclipse.e4.demo.mailapp.mailservice
- org.eclipse.e4.demo.mailapp.mailservice.mock
- org.eclipse.core.runtime

Open the MANIFEST.MF in org.eclipse.e4.demo.mailapp and export the „org.eclipse.e4.demo.mailapp"-package so that it is visible in our test-bundle.

Add a TestAccountView-Class:

```
package org.eclipse.e4.demo.mailapp.test;

import org.eclipse.core.databinding.observable.Realm;
import org.eclipse.e4.demo.mailapp.AccountView;
import org.eclipse.e4.demo.mailapp.mailservice.mock.MailSessionFactoryImpl;
import org.eclipse.jface.databinding.swt.SWTObservables;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class TestAccountView {
  public static void main(String[] args) {
    final Display d = new Display();
    Realm.runWithDefault(SWTObservables.getRealm(d), new Runnable() {

      public void run() {
        Shell shell = new Shell(d);
        shell.setLayout(new FillLayout());
        AccountView view = new AccountView(shell, new MailSessionFactoryImpl());
        view.setUsername("john");
        view.setPassword("doe");
        view.setHost("tomsondev.bestsolution.at");
        view.init();

        shell.open();

        while( !shell.isDisposed() ) {
          if( ! d.readAndDispatch() ) {
            d.sleep();
          }
        }
      }
    });

    d.dispose();
  }
}
```

And launch it as a standard Java-Application to and you should see something like this:

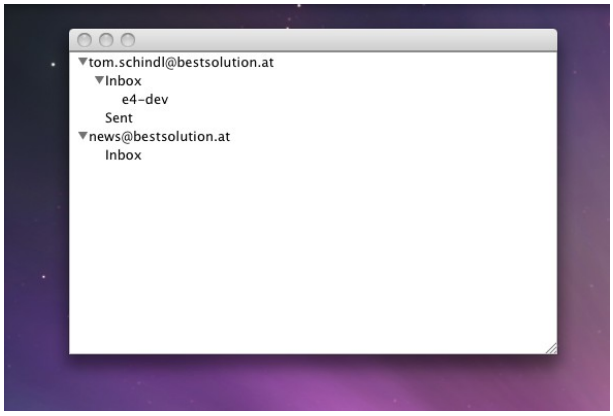


Abbildung 9: Test Account UI

Create the FolderView UI

This view displays all mails of a folder in a SWT-Table. Here's the code and the class to test it.

```
package org.eclipse.e4.demo.mailapp;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import org.eclipse.e4.demo.mailapp.mailservice.domain.IFolder;
import org.eclipse.e4.demo.mailapp.mailservice.domain.IMail;
import org.eclipse.jface.viewers.ArrayContentProvider;
import org.eclipse.jface.viewers.ColumnLabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.jface.viewers.TableViewerColumn;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;

public class FolderView {
  private TableViewer viewer;

  public FolderView(Composite parent) {
    this.viewer = new TableViewer(parent);
    this.viewer.setContentProvider(new ArrayContentProvider());
    this.viewer.getTable().setHeaderVisible(true);
    this.viewer.getTable().setLinesVisible(true);

    TableViewerColumn column = new TableViewerColumn(viewer, SWT.NONE);
    column.getColumn().setText("Subject");
    column.getColumn().setWidth(250);
    column.setLabelProvider(new ColumnLabelProvider() {
      @Override
      public String getText(Object element) {
        return ((IMail)element).getSubject();
      }
    });

    column = new TableViewerColumn(viewer, SWT.NONE);
    column.getColumn().setText("From");
    column.getColumn().setWidth(200);
    column.setLabelProvider(new ColumnLabelProvider() {
```

```
      @Override
      public String getText(Object element) {
        return ((IMail)element).getFrom();
      }
    });

    column = new TableViewerColumn(viewer, SWT.NONE);
    column.getColumn().setText("Date");
    column.getColumn().setWidth(150);
    column.setLabelProvider(new ColumnLabelProvider() {
      private DateFormat format = SimpleDateFormat.getDateTimeInstance();

      @Override
      public String getText(Object element) {
        Date date = ((IMail)element).getDate();
        if( date != null ) {
          return format.format(date);
        }
        return "-";
      }
    });
  }

  public void setFolder(IFolder folder) {
    viewer.setInput(folder.getSession().getMails(folder, 0, folder.getMailCount()));
  }
}
```

And the class to test it:

```
package org.eclipse.e4.demo.mailapp.test;

import org.eclipse.core.databinding.observable.Realm;
import org.eclipse.e4.demo.mailapp.FolderView;
import org.eclipse.e4.demo.mailapp.mailservice.domain.IAccount;
import org.eclipse.e4.demo.mailapp.mailservice.mock.MailSessionFactoryImpl;
import org.eclipse.jface.databinding.swt.SWTObservables;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class TestFolderView {
  public static void main(String[] args) {
    final Display d = new Display();
    Realm.runWithDefault(SWTObservables.getRealm(d), new Runnable() {

      public void run() {
        Shell shell = new Shell(d);
        shell.setLayout(new FillLayout());
        FolderView view = new FolderView(shell);
        view.setFolder(((IAccount)new MailSessionFactoryImpl().openSession("", "john",
"doe").getAccounts().get(0)).getFolders().get(0));

        shell.open();
        while( !shell.isDisposed() ) {
          if( ! d.readAndDispatch() ) {
            d.sleep();
          }
        }
      }
    });
```

```
    d.dispose();
  }
}
```

The UI you should see when running the Java Application looks like this:

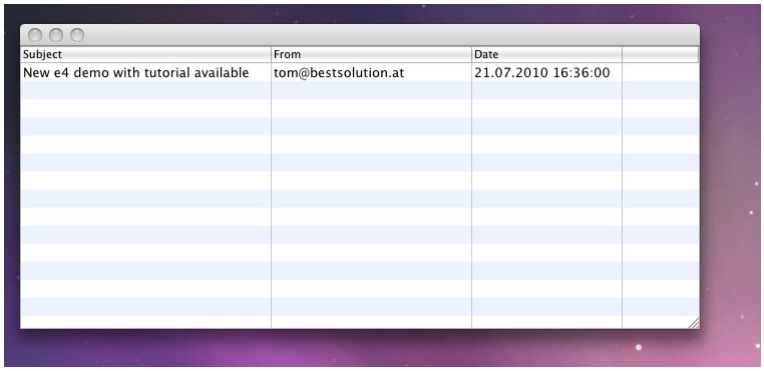


Abbildung 10: Test FolderView

Create the Mail UI

This UI displays the a selected mail to the user. The Java code for the View looks like this:

```
package org.eclipse.e4.demo.mailapp;

import org.eclipse.core.databinding.DataBindingContext;
import org.eclipse.core.databinding.ObservablesManager;
import org.eclipse.core.databinding.beans.BeanProperties;
import org.eclipse.core.databinding.observable.value.WritableValue;
import org.eclipse.e4.demo.mailapp.mailservice.domain.IMail;
import org.eclipse.jface.databinding.swt.WidgetProperties;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;

public class MailView {
  private DataBindingContext dbc;
  private WritableValue mail = new WritableValue();
  private ObservablesManager manager;

  public MailView(final Composite composite) {
    dbc = new DataBindingContext();
    manager = new ObservablesManager();
    manager.runAndCollect(new Runnable() {
      public void run() {
        initUI(composite);
      }
    });
  }

  public void setMail(IMail mail) {
```

```
    if( mail != null ) {
      this.mail.setValue(mail);
    }
  }

  private void initUI(Composite composite) {
    Composite parent = new Composite(composite, SWT.NONE);
    GridLayout gd = new GridLayout();
    gd.horizontalSpacing=0;
    gd.verticalSpacing=0;
    parent.setLayout(gd);

    Composite header = new Composite(parent,SWT.NONE);
    header.setLayout(new GridLayout(2,false));

    Label l = new Label(header, SWT.NONE);
    l.setText("From");

    l = new Label(header, SWT.NONE);
    l.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
    dbc.bindValue(WidgetProperties.text().observe(l),
           BeanProperties.value("from").observeDetail(mail));

    l = new Label(header,SWT.NONE);
    l.setText("Subject");

    l = new Label(header, SWT.NONE);
    l.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
    dbc.bindValue(WidgetProperties.text().observe(l),
           BeanProperties.value("subject").observeDetail(mail));

    l = new Label(header,SWT.NONE);
    l.setText("To");

    l = new Label(header, SWT.NONE);
    l.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
    dbc.bindValue(WidgetProperties.text().observe(l),
           BeanProperties.value("to").observeDetail(mail));

    l = new Label(parent, SWT.SEPARATOR|SWT.HORIZONTAL);
    l.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    Text t = new Text(parent, SWT.BORDER|SWT.V_SCROLL|SWT.H_SCROLL|SWT.WRAP);
    t.setLayoutData(new GridData(GridData.FILL_BOTH));
    t.setEditable(false);
    dbc.bindValue(WidgetProperties.text().observe(t),
           BeanProperties.value("body").observeDetail(mail));
  }

  public void dipose() {
    manager.dispose();
  }
}
```

And the class to test it:

```
package org.eclipse.e4.demo.mailapp.test;

import org.eclipse.core.databinding.observable.Realm;
import org.eclipse.e4.demo.mailapp.MailView;
import org.eclipse.e4.demo.mailapp.mailservice.domain.IAccount;
```

```
import org.eclipse.e4.demo.mailapp.mailservice.domain.IFolder;
import org.eclipse.e4.demo.mailapp.mailservice.mock.MailSessionFactoryImpl;
import org.eclipse.jface.databinding.swt.SWTObservables;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class TestMailView {
  public static void main(String[] args) {
    final Display d = new Display();
    Realm.runWithDefault(SWTObservables.getRealm(d), new Runnable() {

      public void run() {
        Shell shell = new Shell(d);
        shell.setLayout(new FillLayout());
        MailView view = new MailView(shell);
        IFolder folder = ((IAccount)new MailSessionFactoryImpl().openSession("", "john",
"doe").getAccounts().get(0)).getFolders().get(0);

        view.setMail(folder.getSession().getMails(folder, 0, 1).get(0));
        shell.open();

        while( !shell.isDisposed() ) {
          if( ! d.readAndDispatch() ) {
            d.sleep();
          }
        }
      }
    });

    d.dispose();
  }
}
```

Once more running the test-Programm should create an UI like this:

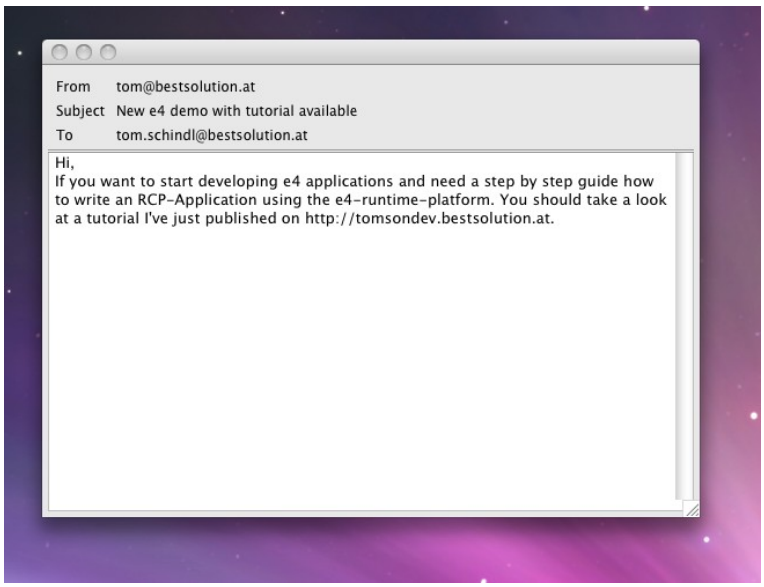


Abbildung 11: Test MailView

We have now created all UI parts of our application without the need to have

any knowledge about the e4-Framework. All we had to know is SWT/JFace and how to programm in Java.

TODO: Provide source code after Chapter 4 as a zip

Assemble an e4-Application

DI and The POJO Application Programming Model

We now have 3 UI-Classes who are on their own are not making much sense but when connected together they are able to build a complete MailReader application.

Before we start with the process of integrating our POJOs in the Application model I think it makes sense to explain what we are going to do in the next view sections of this tutorial.

I assume most of you have already heard at least once about Dependency Injection (DI). Those of you who have ever worked with Spring or Guice are familiar with the concepts for others who never had don't be afraid its not really complex.

Let's take a look at a typical 3.x where are are reacting on the change of the current selection in the workbench.

```
public class View extends ViewPart {


  public void createPartControl(Composite parent) {

    getSite().getSelectionProvider().addSelectionChangedListener(new ISelectionChangedListener() {
      public void selectionChanged(SelectionChangedEvent event) {
        if( event.getSelection() instanceof IStructuredSelection) {
          Object o = ((IstructuredSelection)event.getSelection()).getFirstElement();

          if( o instanceof IFolder ) {
            updateFolder((IFolder) o);

          }

        }

      }

    });

  }

  void updateFolder(IFolder folder) {
    // Do something when selection changes
  }
}
```

And here's what you write in e4

```
public class View {

  @Inject
```

```
  @Named(IServiceConstants.ACTIVE_SELECTION)
  void updateFolder(IFolder folder) {
    // Do something when selection changes
  }
}
```

The first thing you notice is that the code is much more concise and you don't have to write tons of glue code but more important is that you are flipping sides.

Instead of being the active part you get the inactive one who gets informed automatically if something changes you are in need of.

This makes your code much more reuseable because you are not depending on external stuff like the ISelectionService being available.

I'm not going into great detail here now because DI is a very wide area. The imporant thing for us is that we'll have to add annotations like @Inject at various places in our code (constructors, fields, methods) to get informations we need to make up the UI.

There's no other way to get access to informations because e4 doesn't provide statics or singletons like the 3.x platform did!

Before we start adding the annotations to our code we have to add 2 more bundles to our MANIFEST.MF:

- javax.inject
- javax.annotation
- org.eclipse.e4.core.di
- org.eclipse.e4.ui.services

who provide the Annotations we are going to add to our code and some constants.

Modify the AccountView like this:

```
public class AccountView {
  @Inject
  public AccountView(Composite parent, IMailSessionFactory mailSessionFactory) {
    // Unmodified
  }

  @PostConstruct
  public void init() {
    // Unmodified
  }
}
```

Modify the FolderView like this:

```
public class FolderView {
```

```
  @Inject
  public FolderView(Composite parent) {
    // Unmodified
  }

  @Inject
  public void setFolder(@Named(IServiceConstants.ACTIVE_SELECTION) @Optional IFolder folder) {
    // Unmodified
  }
}
```

and the MailView like this:

```
public class MailView {

  @Inject
  public MailView(final Composite composite) {
    // Unmodified
  }

  @Inject
  public void setMail(@Named(IServiceConstants.ACTIVE_SELECTION) @Optional IMail mail) {
    // Unmodified
  }

  @PreDestroy
  public void dipose() {
    // Unmodified
  }
}
```

Let's try to understand the code parts above a bit better. The first thing you need to know is that the instance of creation and destruction is handled by the e4-DI-Container.

At the moment the some code request an instance of e.g. AccountView the DI-Framework search through constructors annotated with @Inject and tries to satisfy the arguments the constructor. The informations need to call the constructor are looked up something called IEclipseContext which you can think of as a of the type Map<String,Object>.

For the AccountView-constructor from above it searches for 2 keys:

- org.eclipse.swt.widget.Composite
- org.eclipse.e4.demo.mailapp.mailservice.IMailSessionFactory

and passes the value found to the constructor.

After having created an instance of the class it search for fields and methods annotated with **@Inject** and looks up the their value.

In contrast to the constructor stuff though it remembers the injected keys and whenever the value connected to the key changes it reinjects the value.

A special thing in this context is the **@Named** useage which allows one to define the key to use when looking up the value (by default the fully qualified class name is used).

The **@Optional** annotation means that if no value is found or the value stored under the key can not be converter to type the system should pass in NULL instead.

The other 2 annotations you see are controling the lifecycle of the an object. Methods annotated with **@PostConstruct** are used called after the object is created and all injections are done (field and method).

The **@PreDestroy** is the opposite. It is called before the object is destroyed by the DI-Container and provides the possibility to clean up resources allocated by the POJO.

Wireing the POJOs into the Application Model

As noted above the e4-runtime is at its heart a DI-Container which controls the whole application and connects bits and pieces to make up a complete application from those small POJOs.

To connect all this informations it uses the Application model we've already used to define the initial layout of our application. Our POJOs from above are now going to get part of the Application model and the application framework knows how to create instances whenever it needs one (e.g. the UI should be created).

a) Open the Application.e4xmi and create a structure like this:

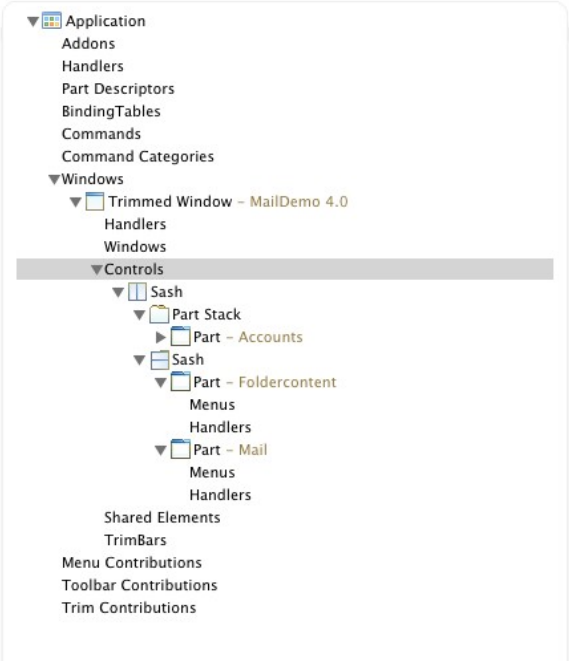


Abbildung 12: UI Model for Mail 4.0

b) Select the 1st Part in the tree and press the „Find...“-button on the Class URI attribute and search for our AccountView-Pojo.

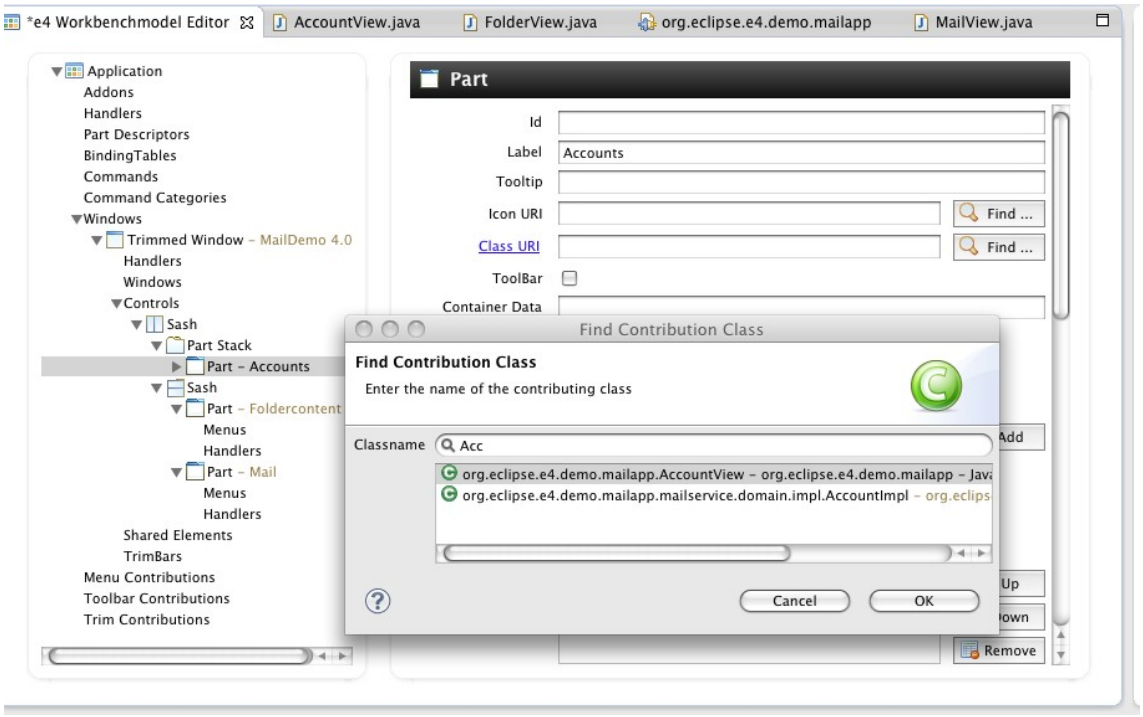


Abbildung 13: Connect POJO with UI

c) Select the 2nd Part in the tree, press the „Find...“-button and select the „FolderView“

d) Select the 3rd in the tree, press the „Find...“-button and select the „MailView“

What we've done in step b) to d) is to wire our UI model with our POJOs and now at the moment the application has to render a part which is connected to such a POJO it creates an instance through the DI-Container and hands over control for this area to the POJO.

When launching our application we should see something like this:

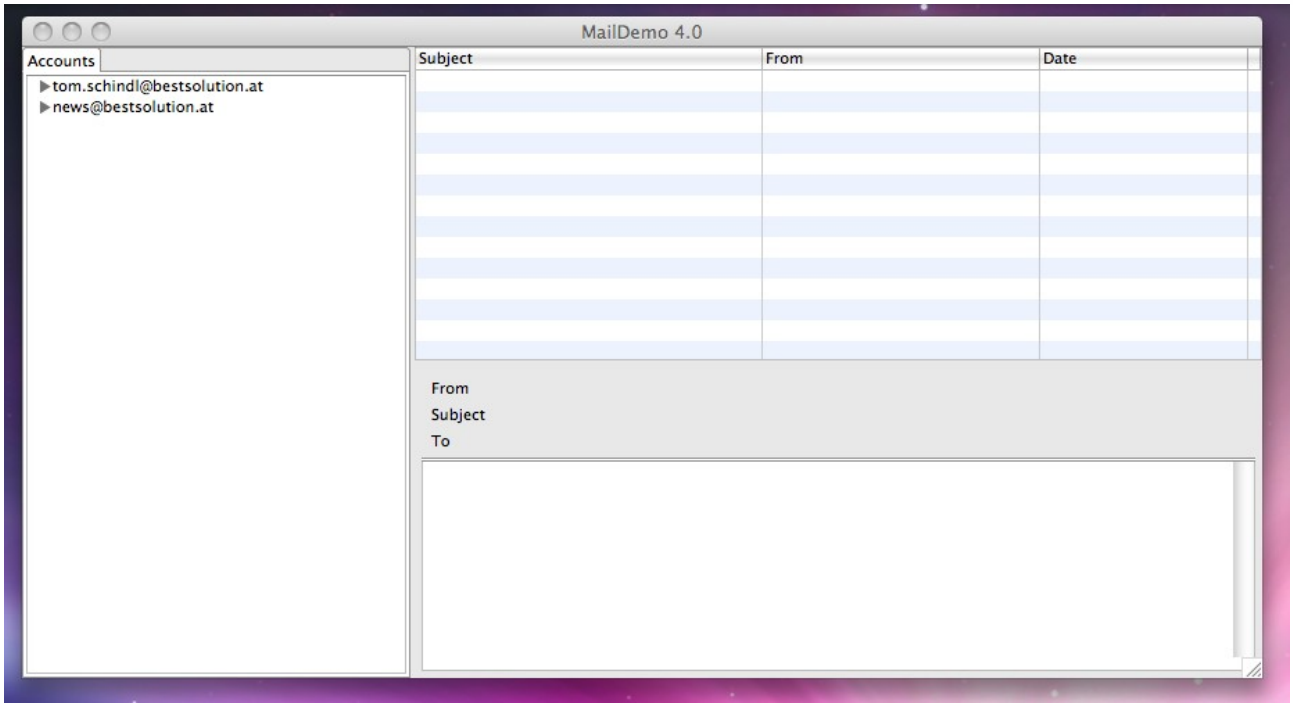


Abbildung 14: Running MailDemo Application

But there's still one thing missing. When selecting an entry in the list the list of mails is not updated. The problem you are seeing here is that the AccountView has to inform others about the changed selection.

This information can be passed around through an special service named ESelectionService. To get access to this service you need to add „org.eclipse.e4.ui.workbench" to your MANIFEST.MF and modify the UI code like this:

AccountView:

```
public class AccountView {
  @Inject
  @Optional
  private ESelectionService selectionService;

  @Inject
  public AccountView(Composite parent, IMailSessionFactory mailSessionFactory) {
    // Unmodified
    viewer.addSelectionChangedListener(new ISelectionChangedListener() {
      public void selectionChanged(SelectionChangedEvent event) {
        if( selectionService != null ) {
          selectionService.setSelection(
            ((IStructuredSelection)event.getSelection()).getFirstElement()
          );
        }
      }
    });
  }

  @PostConstruct
  public void init() {
```

```
    // Unmodified
  }
}
```

FolderView:

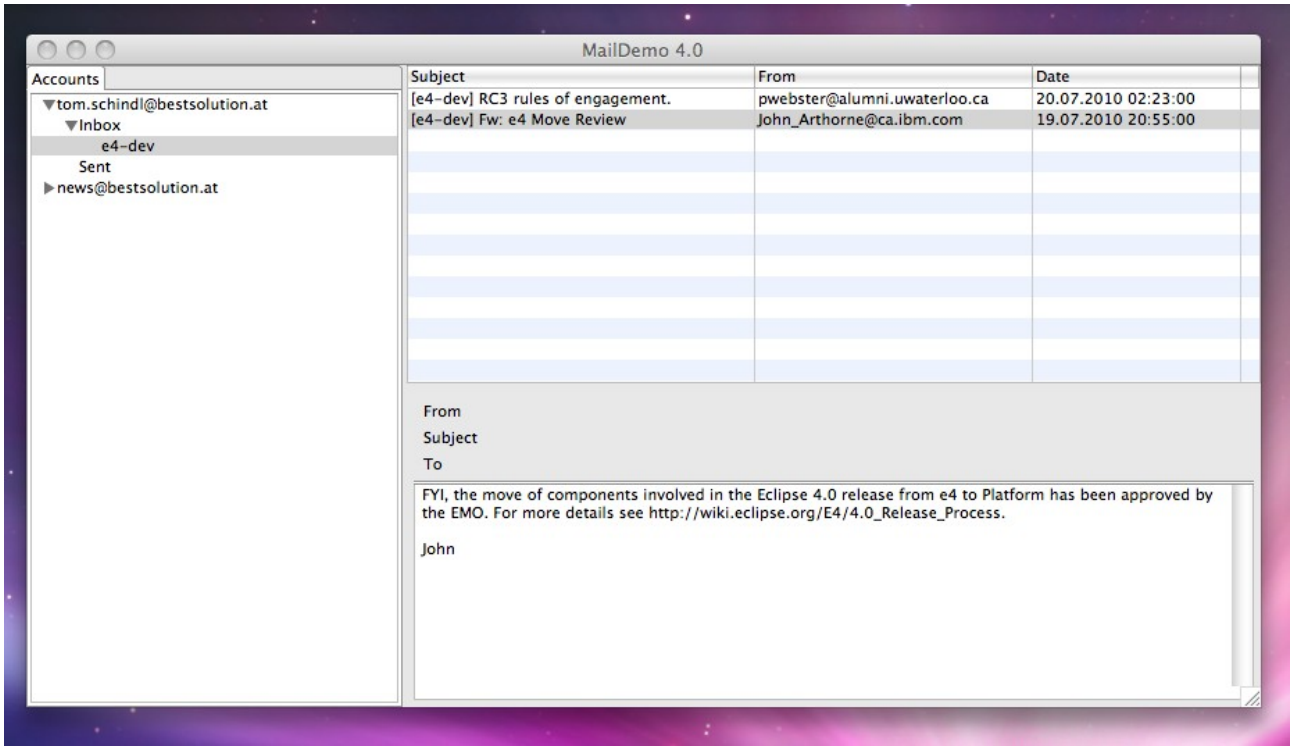
```
public class FolderView {

  @Inject
  @Optional
  private ESelectionService selectionService;


  @Inject
  public FolderView(Composite parent) {
    // Unmodified
    viewer.addSelectionChangedListener(new ISelectionChangedListener() {
      public void selectionChanged(SelectionChangedEvent event) {
        if( selectionService != null ) {
          selectionService.setSelection(
            ((IStructuredSelection)event.getSelection()).getFirstElement()
          );
        }
      }
    });
  }

  @Inject
  public void setFolder(@Named(IServiceConstants.ACTIVE_SELECTION) @Optional IFolder folder) {
    // Unmodified
  }
}
```

The application should now behave as expected and look like this.



TODO: Provide source code after Chapter 5 as a zip

TODO: CSS

Add some CSS stuff

TODO: Preferences

Read username and password from preferences

TODO: Add Event System

Send out an event when a new message arrives

TODO: Contributing Fragments

Contribute a write mail dialog