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.

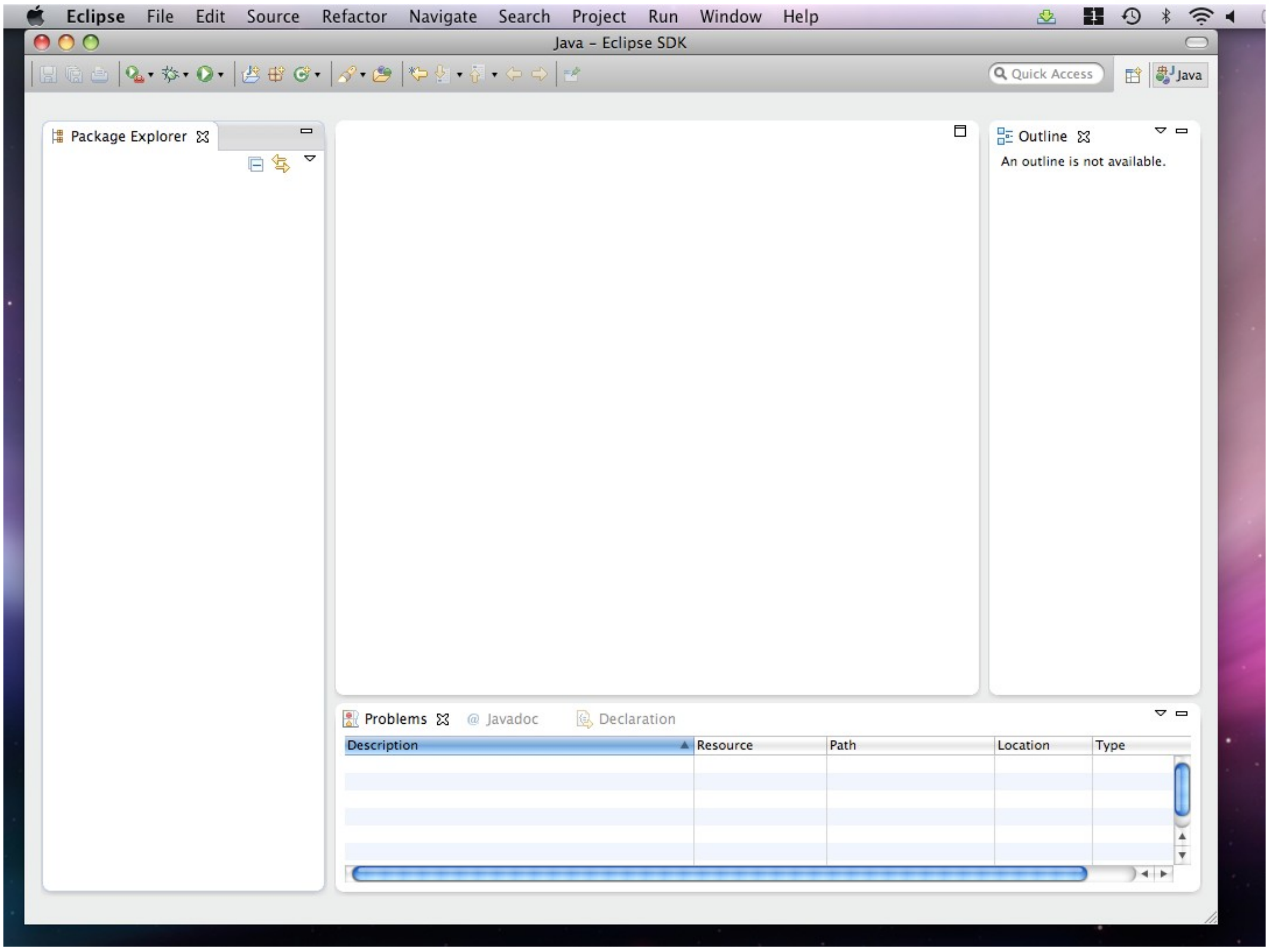
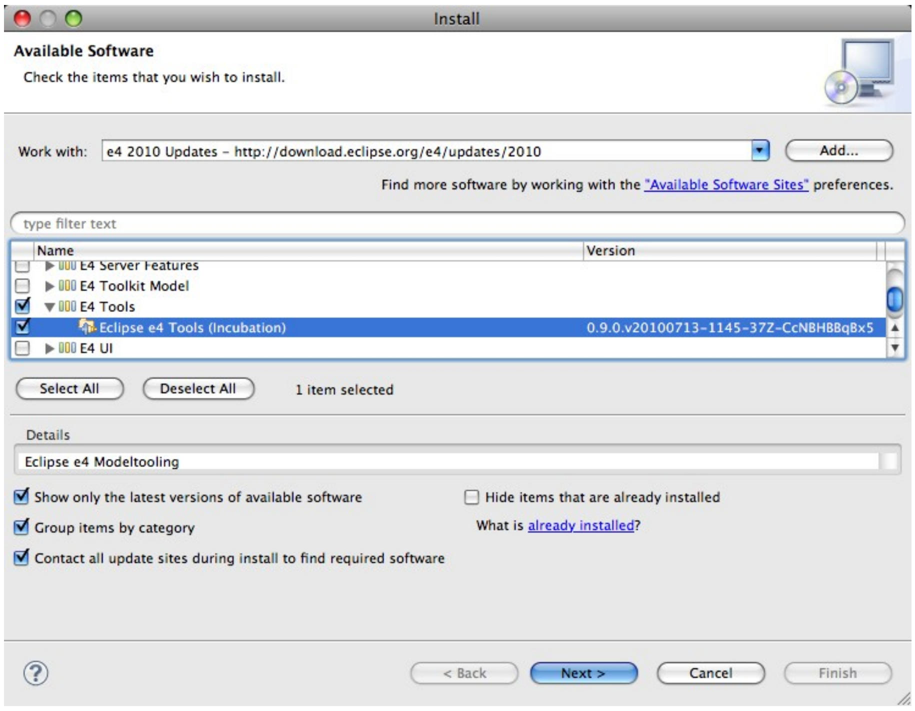


Abbildung 1: 4.0 SDK

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 … .


Install

Available Software

Check the items that you wish to install.

Work with: e4 2010 Updates – http://download.eclipse.org/e4/updates/2010

Find more software by working with the "Available Software Sites" preferences.

Name	Version
☐ E4 Server Features	
☐ E4 Toolkit Model	
✓ E4 Tools	
✓ Eclipse e4 Tools (Incubation)	0.9.0.v20100713-1145-37Z-CcNBHBBqBx5
☐ E4 UI	

1 item selected

Details: Eclipse e4 Modeltooling

✓ Show only the latest versions of available software
✓ Group items by category
✓ Contact all update sites during install to find required software
☐ Hide items that are already installed

What is already installed?

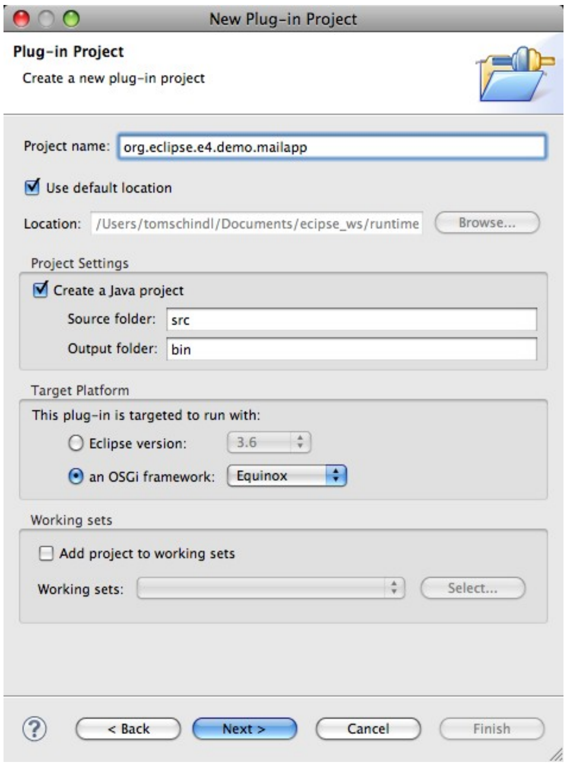
Abbildung 2: Install e4 Tooling

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:


New Plug-in Project

Plug-in Project
Create a new plug-in project

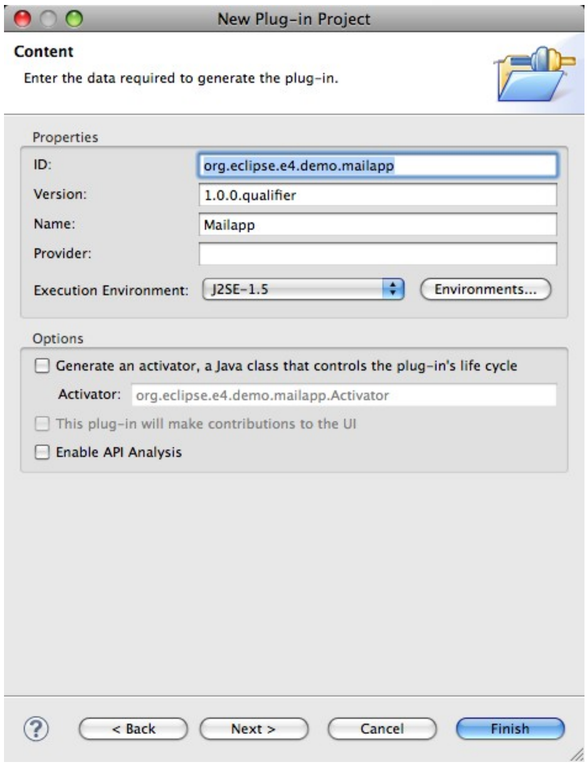
Project name: org.eclipse.e4.demo.mailapp
[✓] Use default location
Location: /Users/tomschindl/Documents/ecipse_ws/runtime

Project Settings
[✓] Create a Java project
Source folder: src
Output folder: bin

Target Platform
This plug-in is targeted to run with:
( ) Eclipse version: 3.6
(●) an OSGi framework: Equinox

Working sets
[ ] Add project to working sets
Working sets:

Abbildung 3: New OSGi-Project 1


New Plug-in Project

Content
Enter the data required to generate the plug-in.

Properties
ID: org.eclipse.e4.demo.mailapp
Version: 1.0.0.qualifier
Name: Mailapp
Provider:
Execution Environment: J2SE-1.5

Options
[ ] Generate an activator, a Java class that controls the plug-in's life cycle
Activator: org.eclipse.e4.demo.mailapp.Activator
[ ] This plug-in will make contributions to the UI
[ ] Enable API Analysis

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>
```

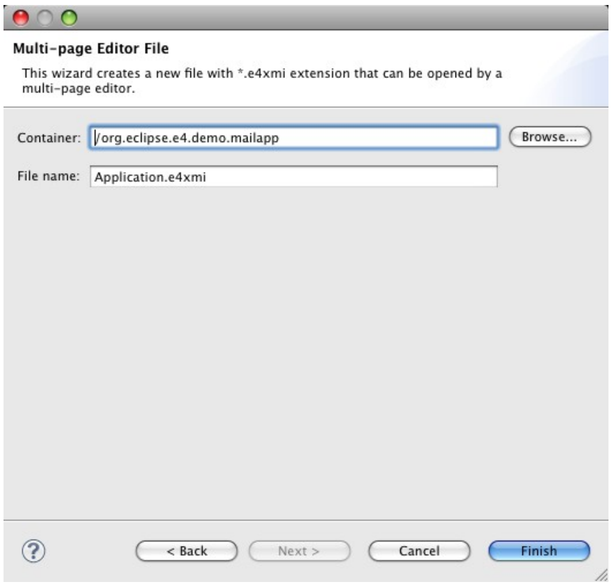
What's missing now to launch our application for the first time are 2 things we need to do. An e4-Application which uses the predefined E4Application has to have a minimal workbench model (we'll learn about this in the next sections), and a .product to define a launchable application.

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:


Multi-page Editor File
This wizard creates a new file with *.e4xmi extension that can be opened by a multi-page editor.
Browse...
< Back
Next >
Cancel
Finish

Container: /org.eclipse.e4.demo.mailapp

File name: Application.e4xmi

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:

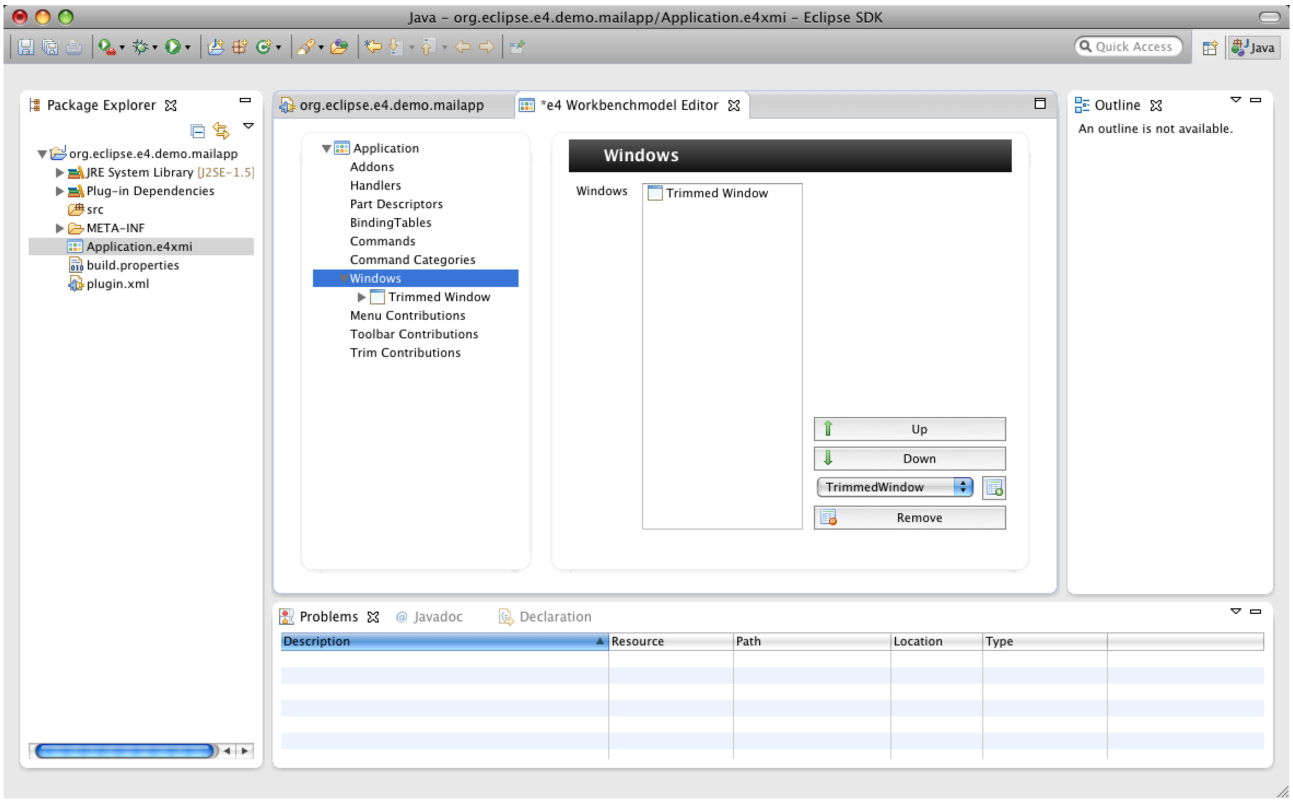


Abbildung 6: Add window to model

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".

Let's take a look at what is written to Application.e4xmi:

```
<?xml version="1.0" encoding="ASCII"?>
<application:Application
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:application="http://www.eclipse.org/ui/2010/UIModel/application"
    xmlns:basic="http://www.eclipse.org/ui/2010/UIModel/application/ui/basic"
    xmi:id="_-K-zoJS1Ed-3RJVy9OYaEA"
    elementId="org.eclipse.e4.demo.mailapp.application">
  <children
      xsi:type="basic:TrimmedWindow"
      xmi:id="_mWWEUJS2Ed-3RJVy9OYaEA"
      label="MailDemo 4.0"
      width="640"
      height="480"/>
</application:Application>
```

You normally don't have to edit this file by hand because e4 provides tooling and I'd also like to point out that XMI is only one possible serialization format of the EMF-Model we just created – yes you've just created your first instance of an EMF-Model.

The default system expects to have a model loaded from such an XMI-File but the framework allows you to replace this through your own model loading/construction strategy if you are not comfortable with the default – all the framework internally cares about is to get an in memory EMF-Model of the application whether loaded from XMI, constructed on the fly, loaded over the wire, … is something totally up to you.

One of the important things you notice in the file are the xmi:id-Attributes who have a very cryptic value which is needed by the default implementation used to restore the application state when started.

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: This adds declarative OSGi-Services who use the extender pattern and so none of the framework has a dependency on it
    - org.eclipse.equinox.event: This provides e4 **the** system it uses and there's only one event system based upon the one provided by OSGi-EventAdmin-service
    - org.eclipse.e4.ui.workbench.renderers.swt: e4 comes with a very flexible rendering system which allows people to completely replace the rendering. We are using the default one provided by e4-Team.
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:

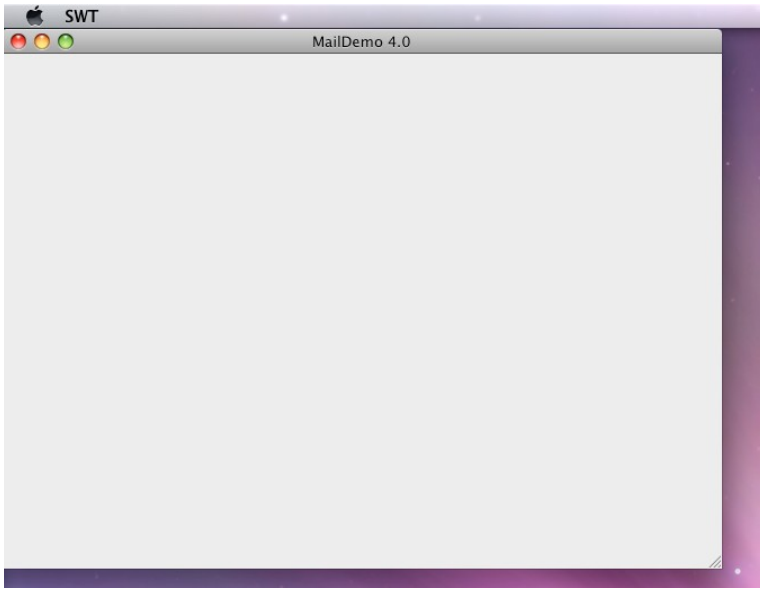


Abbildung 7: First Application

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.

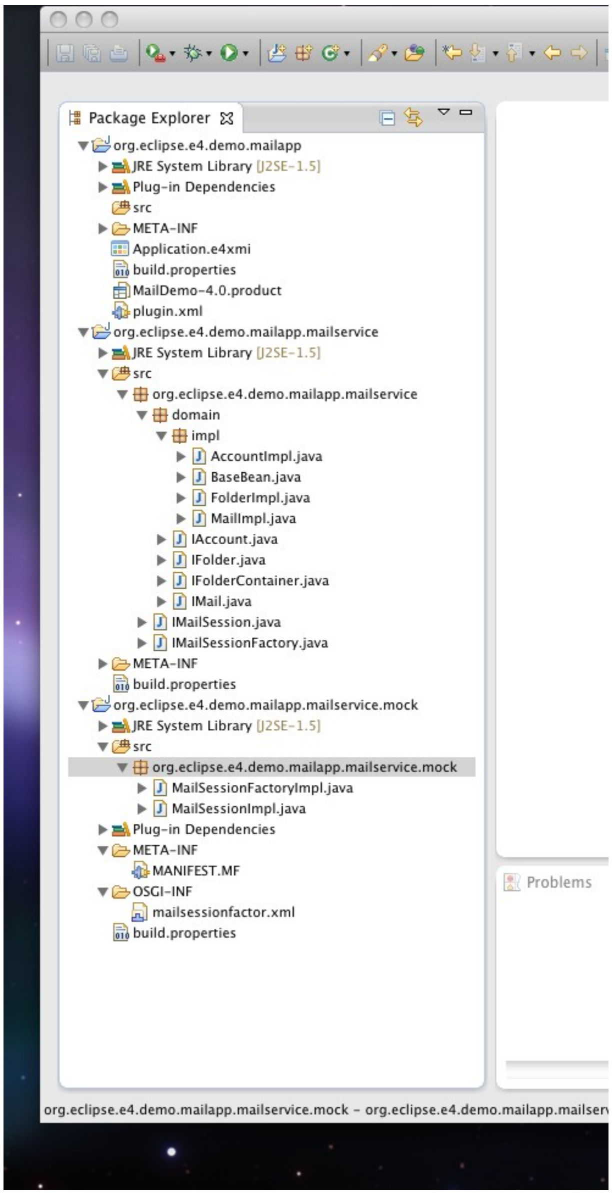
TODO: Location where user can download 2 bundles as a zip

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:



```
Package Explorer
▼ org.eclipse.e4.demo.mailapp
  ▶ JRE System Library [J2SE-1.5]
  ▶ Plug-in Dependencies
    src
  ▶ META-INF
    Application.e4xmi
    build.properties
    MailDemo-4.0.product
    plugin.xml
▼ org.eclipse.e4.demo.mailapp.mailservice
  ▶ JRE System Library [J2SE-1.5]
  ▼ src
    ▼ org.eclipse.e4.demo.mailapp.mailservice
      ▼ domain
        ▼ impl
          ▶ AccountImpl.java
          ▶ BaseBean.java
          ▶ FolderImpl.java
          ▶ MailImpl.java
        ▶ IAccount.java
        ▶ IFolder.java
        ▶ IFolderContainer.java
        ▶ IMail.java
      ▶ IMailSession.java
      ▶ IMailSessionFactory.java
  ▶ META-INF
    build.properties
▼ org.eclipse.e4.demo.mailapp.mailservice.mock
  ▶ JRE System Library [J2SE-1.5]
  ▼ src
    ▼ org.eclipse.e4.demo.mailapp.mailservice.mock
      ▶ MailSessionFactoryImpl.java
      ▶ MailSessionImpl.java
  ▶ Plug-in Dependencies
  ▼ META-INF
      MANIFEST.MF
  ▼ OSGI-INF
      mailsessionfactor.xml
    build.properties
```

Problems

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

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 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.

```
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,SWT.FULL_SELECTION);
    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 doing.

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:

```java
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:

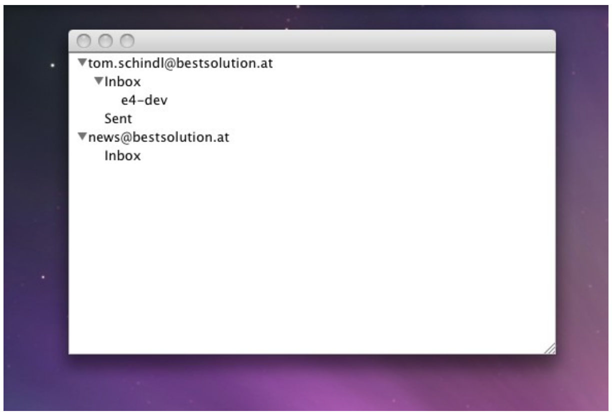


Abbildung 9: Test Account UI

Create the FolderView UI

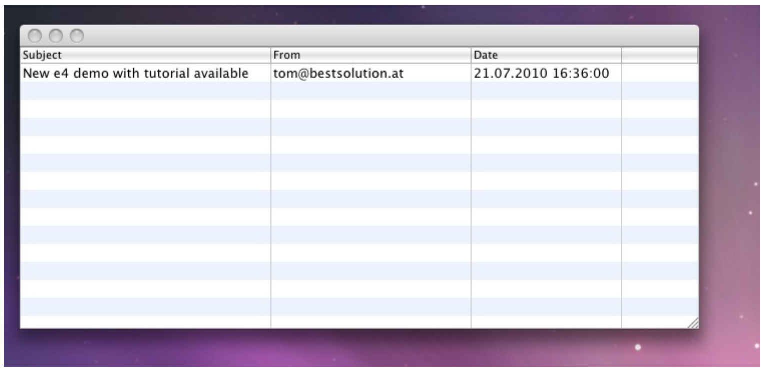
This view displays all mails of a folder in a SWT-Table. Here's the code and the class to test it.

```
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:

```
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:



Subject	From	Date
New e4 demo with tutorial available	tom@bestsolution.at	21.07.2010 16:36:00

Abbildung 10: Test FolderView

Create the Mail UI

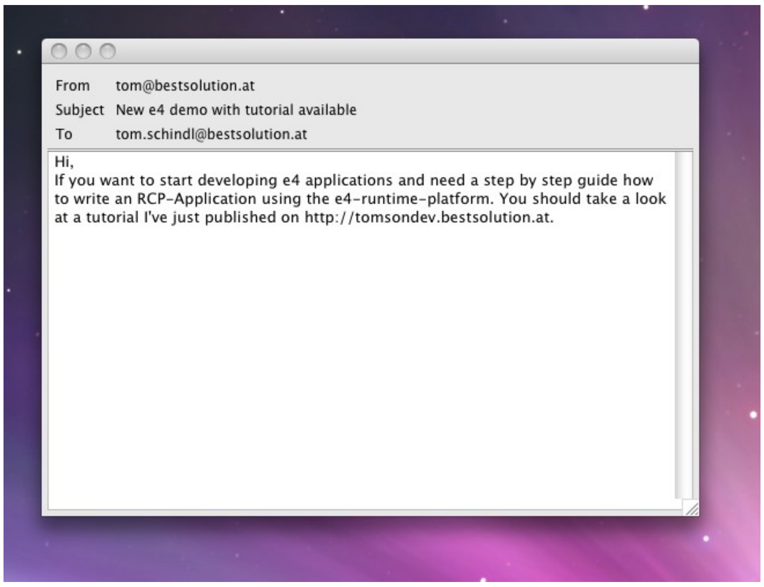
This UI displays the a selected mail to the user. The Java code for the View looks like this:

```
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));
    header.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    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:

```
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:


From tom@bestsolution.at
Subject New e4 demo with tutorial available
To tom.schindl@bestsolution.at

Hi,
If you want to start developing e4 applications and need a step by step guide how to write an RCP-Application using the e4-runtime-platform. You should take a look at a tutorial I've just published on http://tomsondev.bestsolution.at.

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 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 code where we 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 some 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 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 of the constructor. The informations need to call the constructor are looked up something called IEclipseContext which you can think of as a Map of 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 converted 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 called after the object is created and all injections are done (field and method).

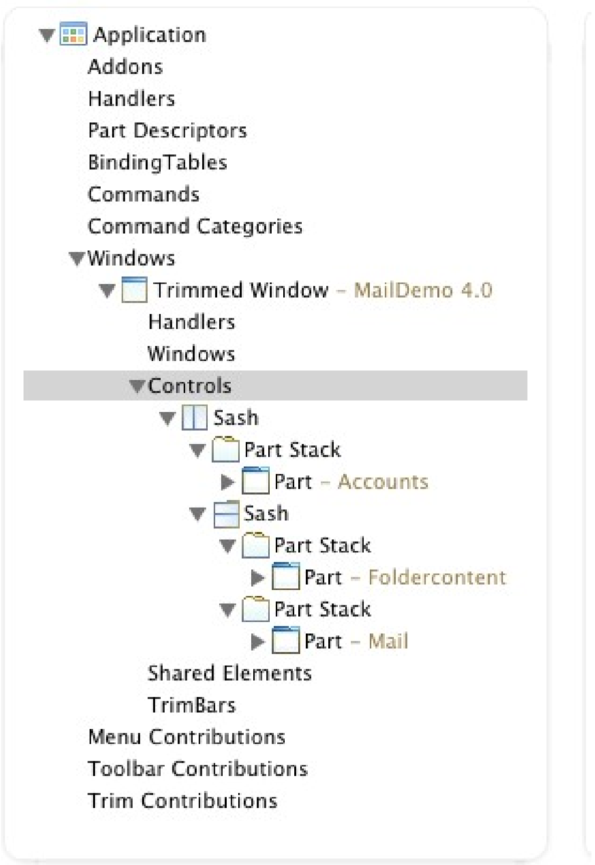
The **@PreDestroy** is the opposite. It is called before the object is destroyed by the DI-Container and provides the possibility to clean up resources allocated by the POJO.

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 because of the DI-Annotations we added the application framework knows how to create instances whenever it needs one.

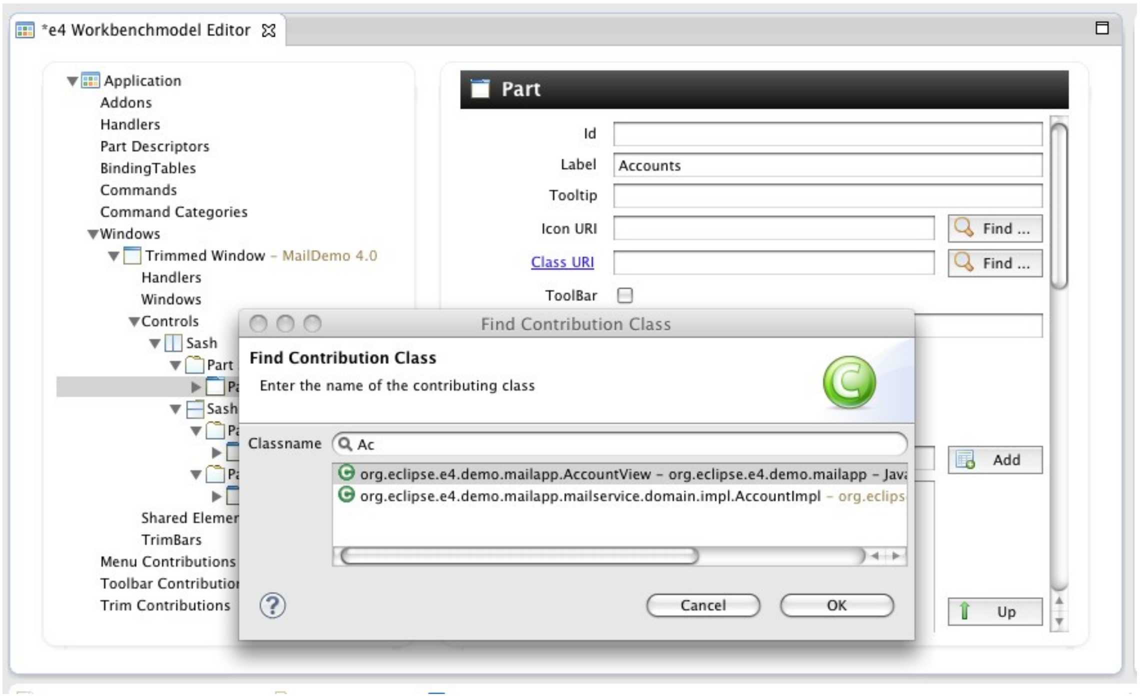
a) Open the Application.e4xmi and create a structure like this:



```
▼ Application
    Addons
    Handlers
    Part Descriptors
    BindingTables
    Commands
    Command Categories
  ▼ Windows
    ▼ Trimmed Window – MailDemo 4.0
        Handlers
        Windows
      ▼ Controls
        ▼ Sash
          ▼ Part Stack
            ▶ Part – Accounts
          ▼ Sash
            ▼ Part Stack
              ▶ Part – Foldercontent
            ▼ Part Stack
              ▶ Part – Mail
        Shared Elements
        TrimBars
    Menu Contributions
    Toolbar Contributions
    Trim Contributions
```

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.



Label: Accounts

Find Contribution Class

Enter the name of the contributing class

Classname: Ac

org.eclipse.e4.demo.mailapp.AccountView – org.eclipse.e4.demo.mailapp – Java
org.eclipse.e4.demo.mailapp.mailservice.domain.impl.AccountImpl – org.eclips

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:

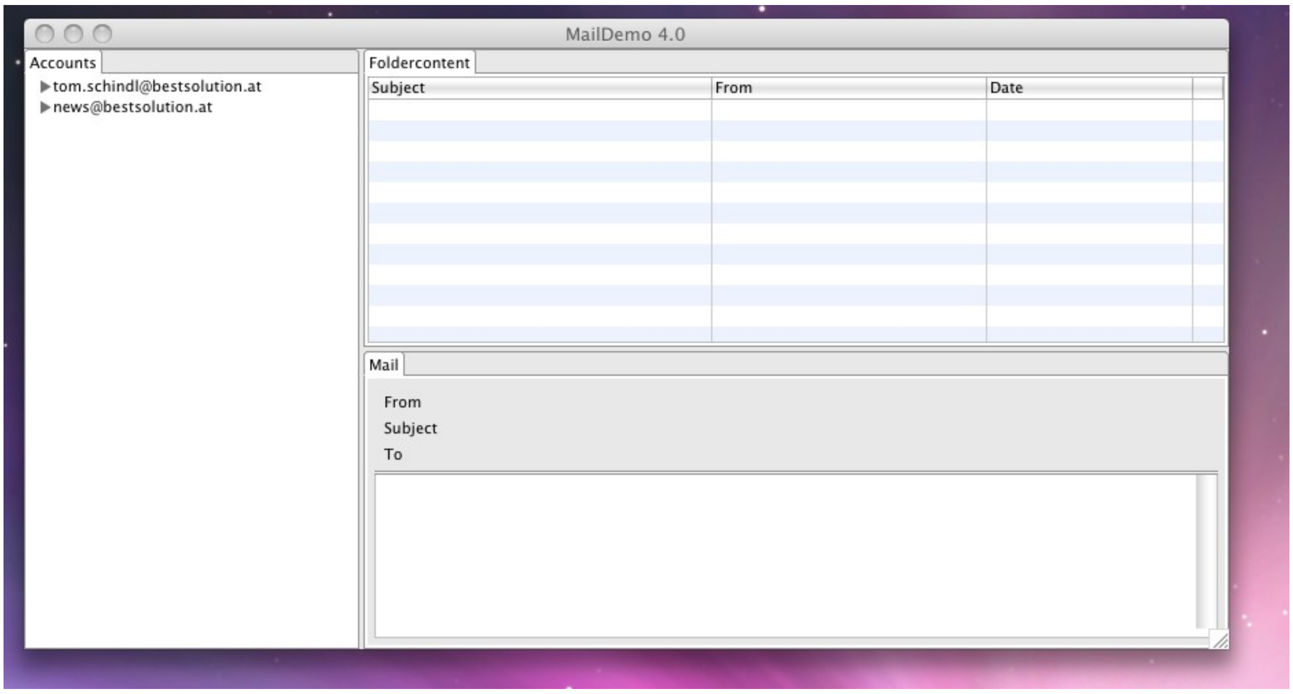


Abbildung 14: Running MailDemo Application

But there's still one thing missing. When selecting an entry in the account area 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:

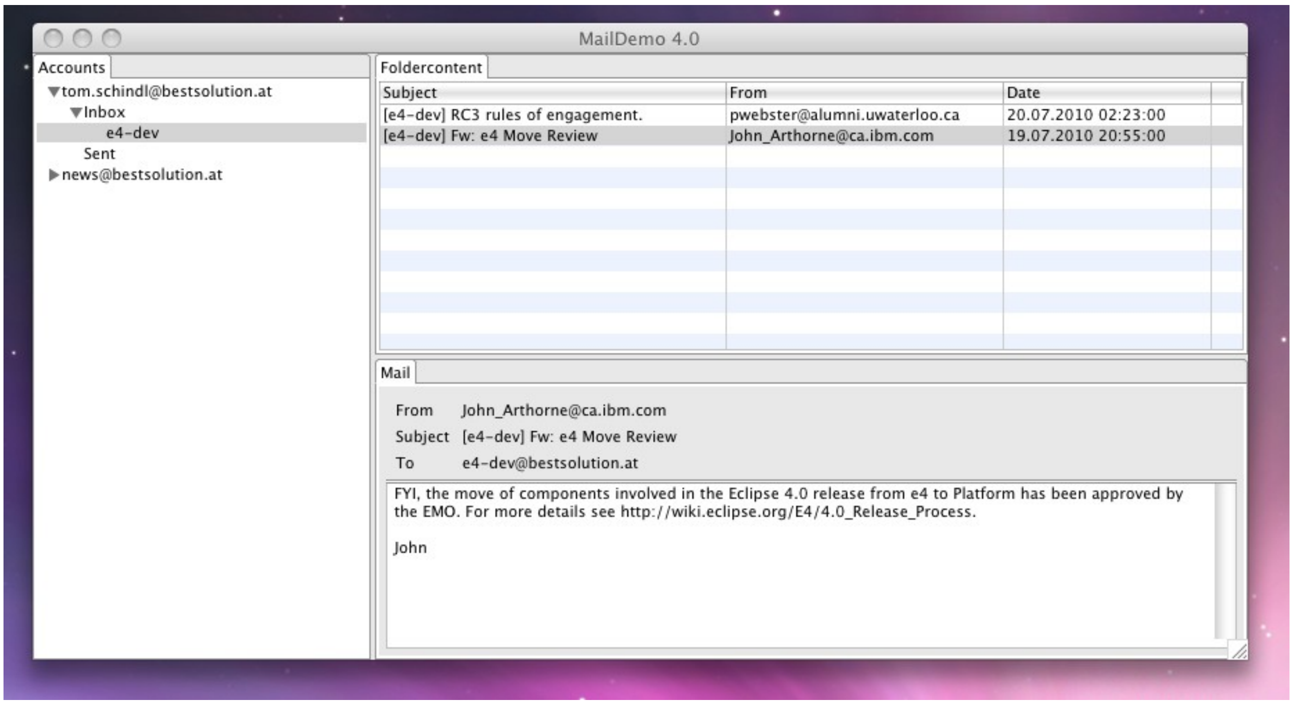
AccountView:

```
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.



Subject	From	Date
[e4-dev] RC3 rules of engagement.	pwebster@alumni.uwaterloo.ca	20.07.2010 02:23:00
[e4-dev] Fw: e4 Move Review	John_Arthorne@ca.ibm.com	19.07.2010 20:55:00

From John_Arthorne@ca.ibm.com
Subject [e4-dev] Fw: e4 Move Review
To e4-dev@bestsolution.at

FYI, the move of components involved in the Eclipse 4.0 release from e4 to Platform has been approved by the EMO. For more details see http://wiki.eclipse.org/E4/4.0_Release_Process.

John

Abbildung 15: Running MailDemo Application after chapter 5

TODO: Provide source code after Chapter 5 as a zip

Improve the Application L&F

Now that we have a running application we can work on a more modern look and feel.

To customize the L&F of applications e4 provides us with CSS-Like declarative syntax. To inform the framework about the CSS-stylesheet it apply to the application we need to:

a) Create a directory css

b) Add a file called default.css

c) Modify the plugin.xml 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>
         <property
               name="applicationCSS"
               value="platform:/plugin/org.eclipse.e4.demo.mailapp/css/default.css">
         </property>
      </product>
   </extension>

</plugin>
```

The applicationCSS-property informs the system about the fact that there's a CSS-stylesheet which has to be applied on the whole RCP-Application.

That's all needed to configure your application to use CSS informations all we now have to do is add informations to the default.css to tweak the L&F.

Here's the initial CSS-Information we are adding:

```
.MTrimmedWindow {
    background-color: #E8E8E8;
    margin-top: 10px;
    margin-bottom: 2px;
    margin-left: 5px;
    margin-right: 5px;
}
```

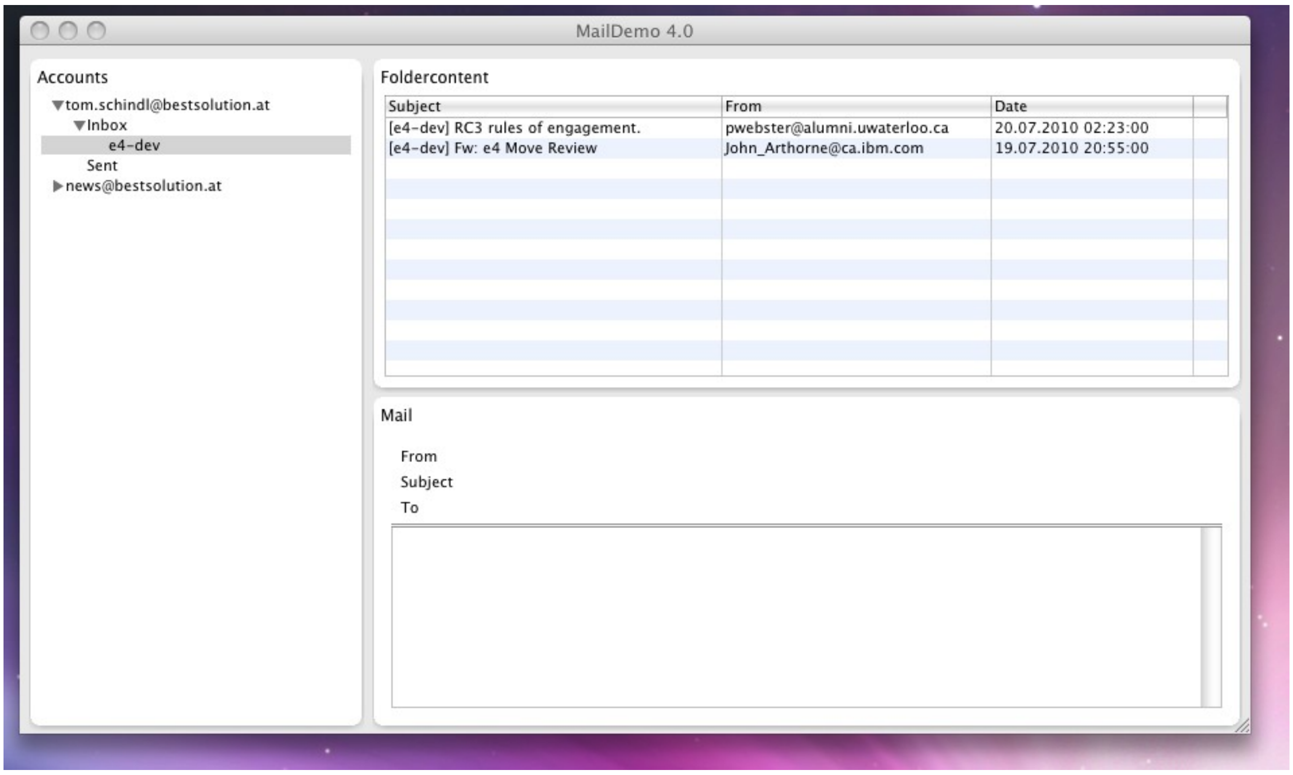
```
.MPartStack {
  tab-renderer:
url('platform:/plugin/org.eclipse.e4.ui.workbench.renderers.swt/org.eclipse.e4.ui.workbench.rendere
rs.swt.CTabRendering');
  unselected-tabs-color: #FFFFFF #FFFFFF #FFFFFF 100% 100%;
  outer-keyline-color: #FFFFFF;
  inner-keyline-color: #FFFFFF;
  font-size: 12;
}
```

What you see above is the definition of 2 CSS-Classes MTrimmedWindow and MPartStack. The name of those classes are the ones of the Application-Model elements we are used prefixed with an "M".

I think I don't have to explain in great detail the attributes defined in MTrimmedWindow the only probably remarkable thing is that the margin information is not interpreted by the SWT-Widget but needs programmatic intervention by the programmer updating the layout.

The informations on MPartStack are more interesting because they differ from what we know from the Web.

The most interesting one is the tab-renderer-attribute which is pointing to a Java class which can be set since 3.6 on a CTabFolder to influence how it is drawn. To find out what the others are doing I'd suggest you play around to see what their effect is.

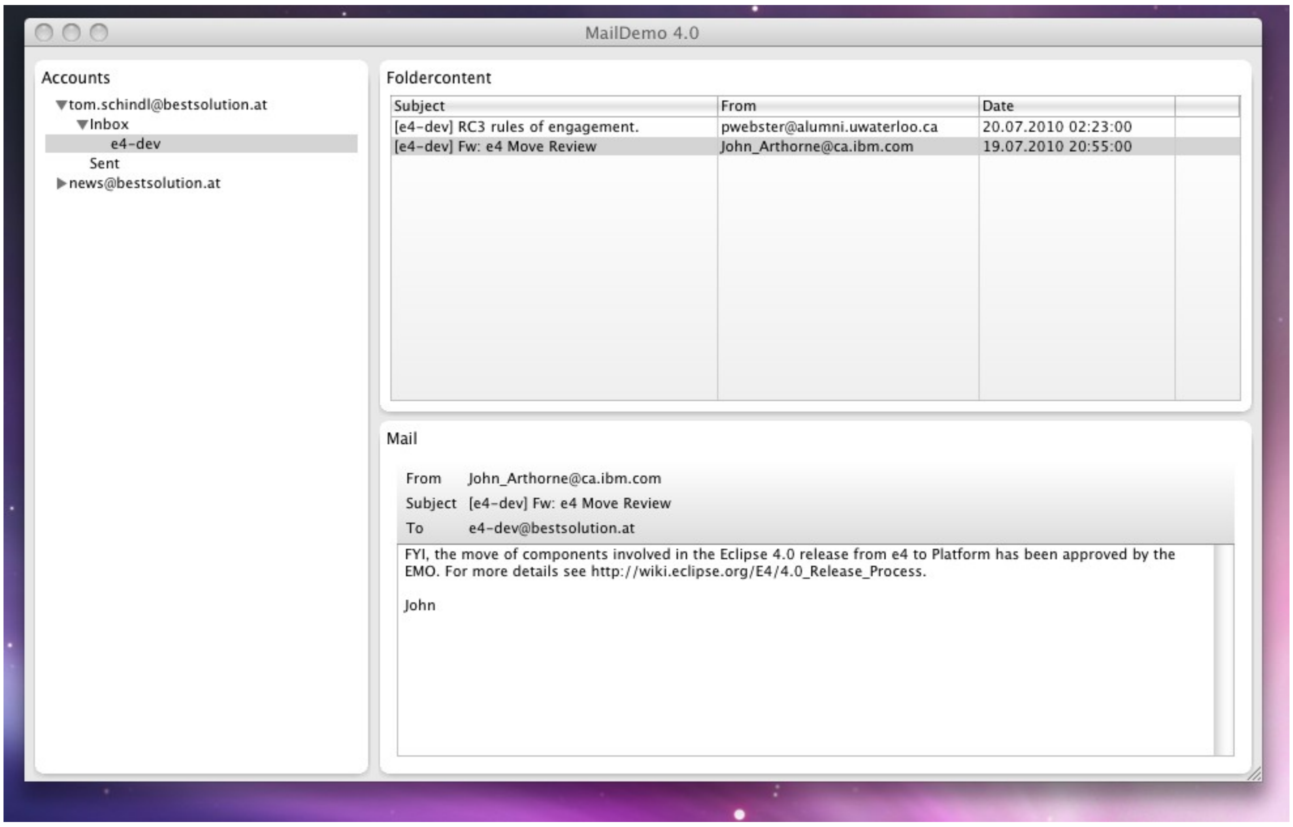


▼tom.schindl@bestsolution.at
  ▼Inbox
    e4-dev
  Sent
▶news@bestsolution.at

Subject	From	Date
[e4-dev] RC3 rules of engagement.	pwebster@alumni.uwaterloo.ca	20.07.2010 02:23:00
[e4-dev] Fw: e4 Move Review	John_Arthorne@ca.ibm.com	19.07.2010 20:55:00

Abbildung 16: MailDemo with CSS

The first set of style informations have been applied to widget owned and controled by the e4-application framework but we also like to apply css informations on our UI code inside our POJOs.



Subject	From	Date
[e4-dev] RC3 rules of engagement.	pwebster@alumni.uwaterloo.ca	20.07.2010 02:23:00
[e4-dev] Fw: e4 Move Review	John_Arthorne@ca.ibm.com	19.07.2010 20:55:00

From John_Arthorne@ca.ibm.com
Subject [e4-dev] Fw: e4 Move Review
To e4-dev@bestsolution.at

FYI, the move of components involved in the Eclipse 4.0 release from e4 to Platform has been approved by the EMO. For more details see http://wiki.eclipse.org/E4/4.0_Release_Process.

John

Abbildung 17: CSS in custom area

The CSS informations added to the file are:

```
.mailList {
      background-color: #FFF #EEE 100%;
}

.mailHeader {
      background-color: #FFF #DDD 100%
}
```

But to take effect in our application we need to modify our our UI-Code to mark the widget with the CSS-Classnames.

```
public class FolderView {
  @Inject
  public FolderView(Composite parent, @Optional IStylingEngine styleEngine) {
    //Unmodified
    if( styleEngine != null ) {
      styleEngine.setClassname(this.viewer.getControl(), "mailList");
    }
  }
}
```

```
public class MailView {
  @Inject
  public MailView(final Composite composite, @Optional final IStylingEngine styleingEngine) {
    //Unmodified
    manager.runAndCollect(new Runnable() {
     public void run() {
       initUI(composite, styleingEngine);
     }
   });
  }

  private void initUI(Composite composite, IStylingEngine styleingEngine) {
    //Unmodified
    if( styleingEngine != null ) {
       styleingEngine.setClassname(header, "mailHeader");
    }
  }
}
```

There's not much magic, we simply inform the DI-Container that we need another service (IStylingEngine) which we can use to set a class on the widget.

Extended Annotations

@Preference

The next area we are going to take a look at is how we are dealing with preferences. A perfect example for preferences is the username, password and host information used to create a MailSession in AccountView.

```
public class AccountView {
  // ...
  private String username = "john";
  private String password = "doe";
  private String host = "tomsondev.bestsolution.com";

  // ...

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

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

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

To get access to values stored in the preferences e4 provides a special annotation you can use in conjunction with **@Inject** named **@Preference** which can take 2 parameters:

- nodepath (optional): the path to the preference node, by default the bundle name is used

- value: the value key

```
public class AccountView {

  // ...

  private boolean modified = false;

  // ...

  @Inject
  public void setUsername(@Preference("username") String username) {
    this.username = username;
  }

  @Inject
  public void setPassword(@Preference("password") String password) {
    this.password = password;
  }

  @Inject
  public void setHost(@Preference("host") String host) {
    this.host = host;
  }

  @PostConstruct
  public void init() {
    if( username != null && password != null && host != null ) {
      mailSession = mailSessionFactory.openSession(host, username, password);
      if( mailSession != null ) {
        viewer.setInput(mailSession.getAccounts());
      } else {
        viewer.setInput(new WritableList());
      }
    }
    modified = false;
  }

}
```

After having added this code the preferences get injected into our view but we need to react on the changes and inform the user that he probably wants to recreate the mail session.

A simple solution for now is to remember that values have been modified and the next time the AccountView receives the focus ask the user whether he'd like to reconnect.

@Focus

To inform the framework which method we'd like it to call when the view receives the focus all we need to do is to annotate some method in our code using the **@Focus** which is coming from “org.eclipse.e4.ui.di” which you should add to your MANIFEST.MF and add a method like this.

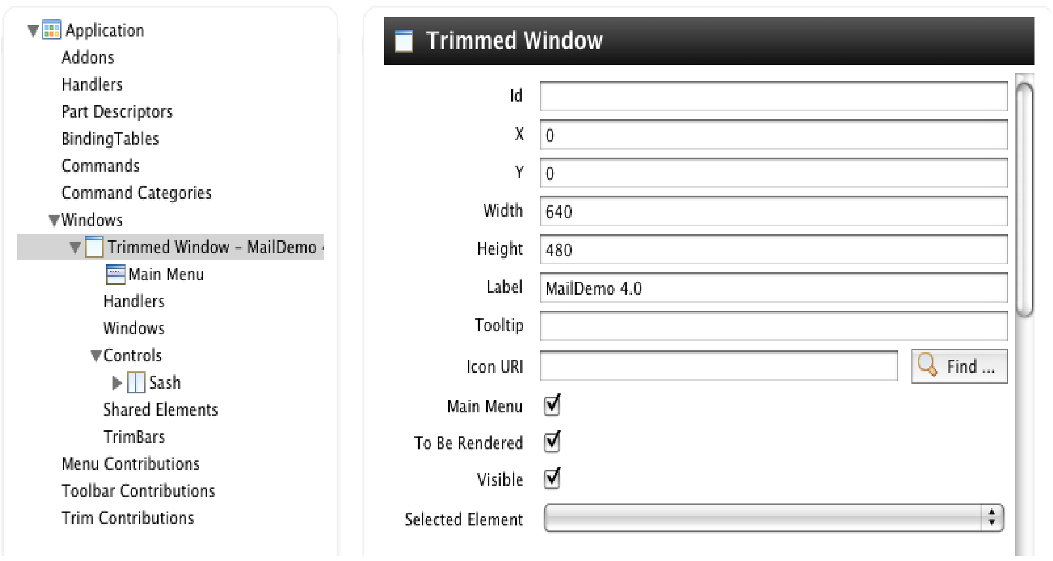
```
@Focus
void onFocus(@Named(IServiceConstants.ACTIVE_SHELL) Shell shell) {
  if( modified ) {
    if( MessageDialog.openQuestion(shell,
           "AccountInfos Modified",
           "The account informations have been modified would you like to reconnect with them?")
    ) {
      init();
      if( mailSession == null ) {
        MessageDialog.openWarning(shell,
             "Connection failed",
             "Opening a connecting to the mail server failed.");
      }
    }
  }
}
```

Menus and @Execute

In the previous chapter we added Preference Support to our AccountView we'll now add a Dialog we use to edit those preferences.

a) Open the Application.e4xmi

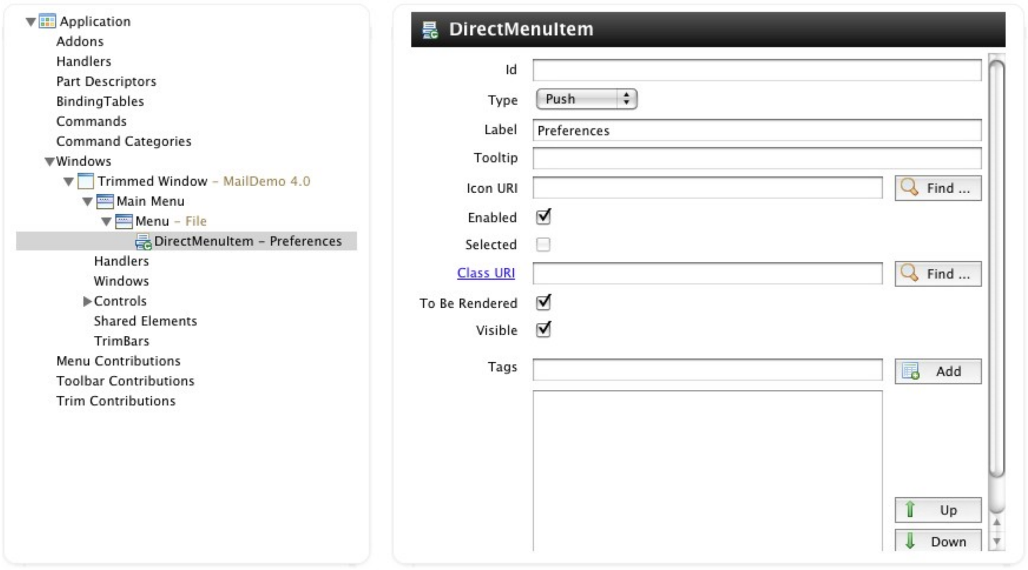
b) Select the Trimmed Window in the Tree and select the “Main Menu” checkbox.



Id	
X	0
Y	0
Width	640
Height	480
Label	MailDemo 4.0
Tooltip	
Icon URI	
Main Menu	✓
To Be Rendered	✓
Visible	✓
Selected Element	

Abbildung 18: Add main menu

c) Create a structure like this



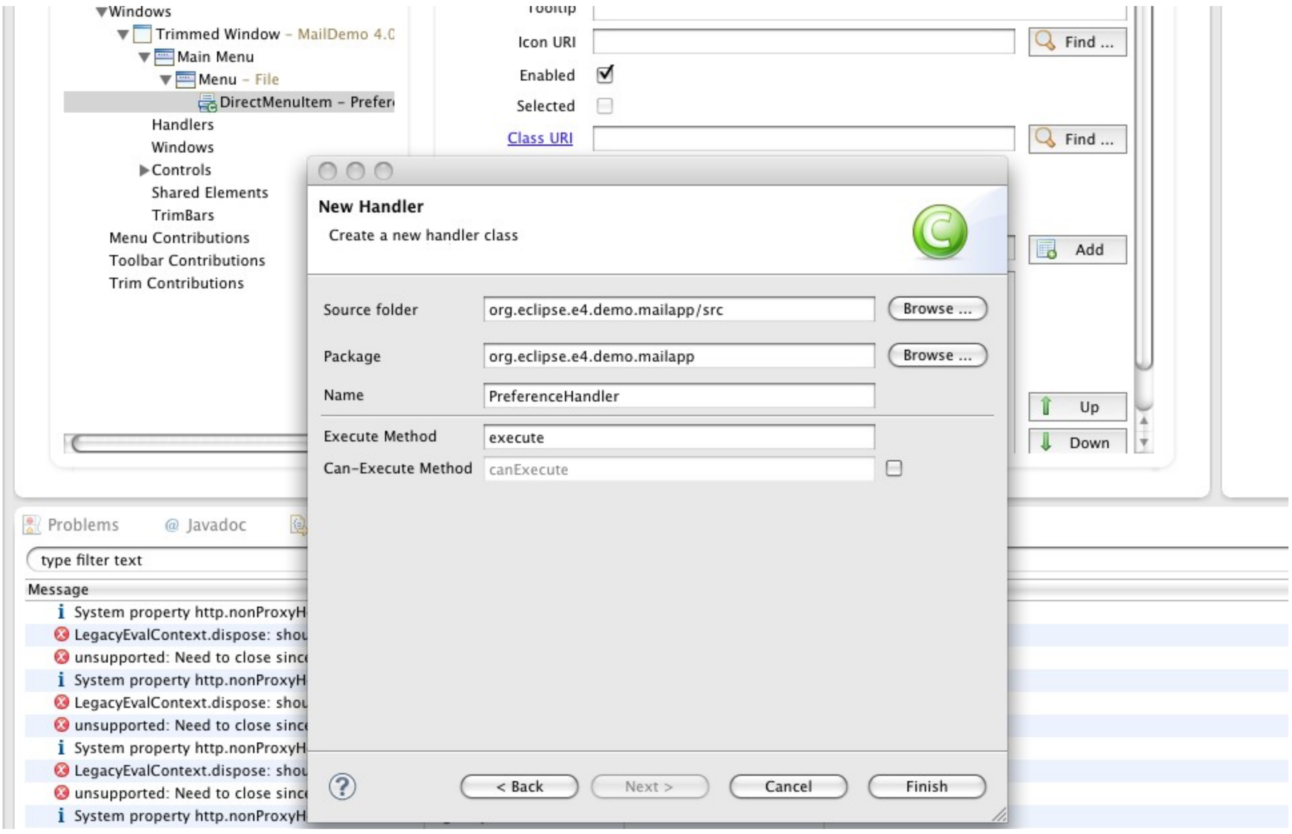
Application
Addons
Handlers
Part Descriptors
BindingTables
Commands
Command Categories
Windows
Trimmed Window – MailDemo 4.0
Main Menu
Menu – File
DirectMenuItem – Preferences
Handlers
Windows
Controls
Shared Elements
TrimBars
Menu Contributions
Toolbar Contributions
Trim Contributions

DirectMenuItem

Id	
Type	Push
Label	Preferences
Tooltip	
Icon URI	
Enabled	✓
Selected	
Class URI	
To Be Rendered	✓
Visible	✓
Tags	

Abbildung 19: Create menustructure

d) Click on the Class URI and enter the following informations



New Handler
Create a new handler class

Source folder	org.eclipse.e4.demo.mailapp/src
Package	org.eclipse.e4.demo.mailapp
Name	PreferenceHandler
Execute Method	execute
Can-Execute Method	canExecute

Abbildung 20: Create handler class

di)

This will generate a class like this:

```
public class PreferenceHandler {
  @Execute
  public void execute() {
  }
}
```

In the above steps we've connected the MenuItem directly to a Java-Class and at the moment the user selects the MenuItem the framework calls the method annotated with **@Execute**.

Before going on to implement the Dialog and the Handler you need to add the following bundles to your MANIFEST.MF:

- org.eclipse.e4.core.contexts
- org.eclipse.equinox.preferences
- org.eclipse.equinox.common

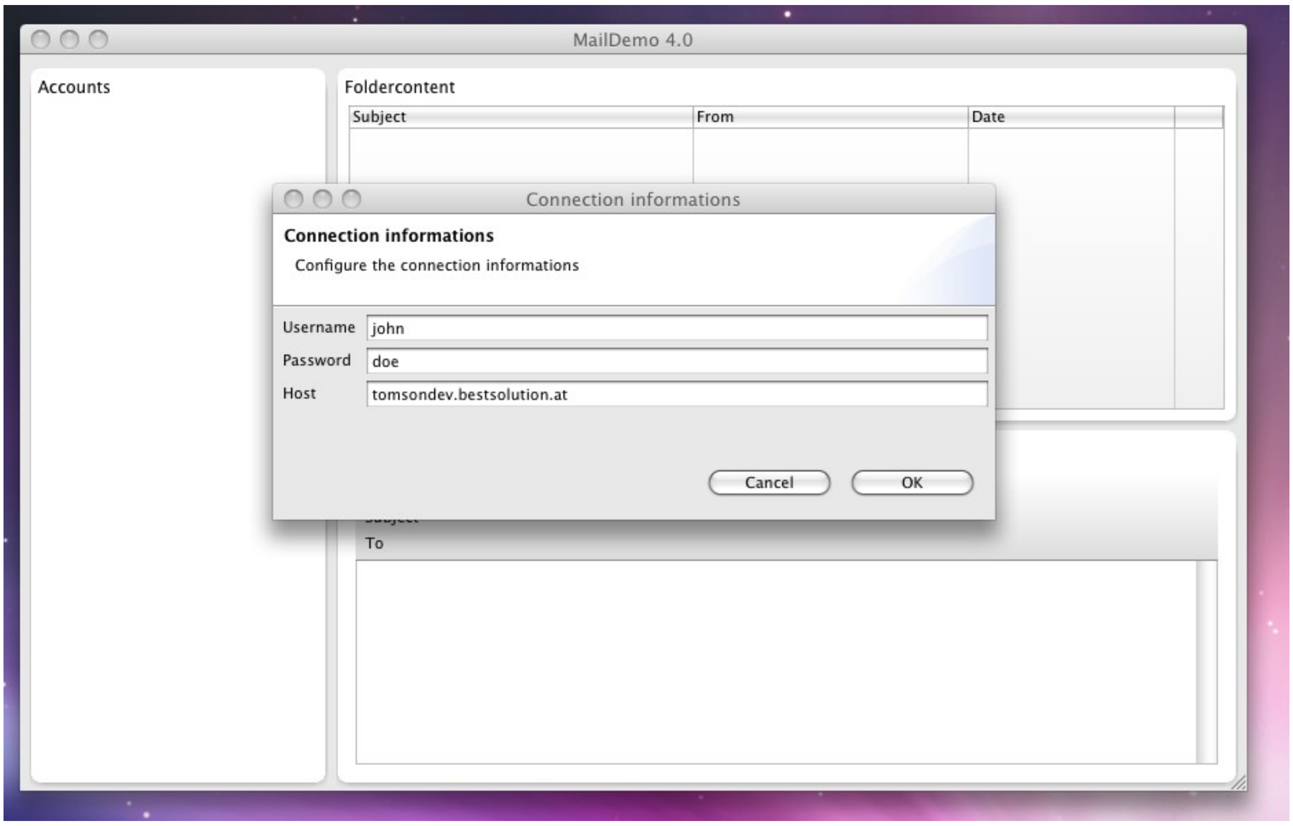
The implementation of the Dialog looks like this:

```
public class PreferenceDialog extends TitleAreaDialog {

  @Inject
  @Preference("username")
  private String username;

  @Inject
  @Preference("password")
  private String password;

  @Inject
  @Preference("host")
  private String host;

  private Text usernameField;
  private Text passwordField;
  private Text hostField;

  @Inject
  public PreferenceDialog(@Named(IServiceConstants.ACTIVE_SHELL) Shell parentShell) {
    super(parentShell);
  }

  @Override
  protected Control createDialogArea(Composite parent) {
    Composite area = (Composite) super.createDialogArea(parent);

    getShell().setText("Connection informations");
    setTitle("Connection informations");
    setMessage("Configure the connection informations");

    Composite container = new Composite(area, SWT.NONE);
    container.setLayoutData(new GridData(GridData.FILL_BOTH));
    container.setLayout(new GridLayout(2, false));

    Label l = new Label(container, SWT.NONE);
    l.setText("Username");

    usernameField = new Text(container, SWT.BORDER);
    usernameField.setText(username == null ? "" : username);
    usernameField.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    l = new Label(container, SWT.NONE);
    l.setText("Password");

    passwordField = new Text(container, SWT.BORDER);
    passwordField.setText(password == null ? "" : password);
    passwordField.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    l = new Label(container, SWT.NONE);
    l.setText("Host");

    hostField = new Text(container, SWT.BORDER);
    hostField.setText(host == null ? "" : host);
    hostField.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    return area;
  }

  @Override
  protected void okPressed() {
    IEclipsePreferences prefs = new InstanceScope().getNode("org.eclipse.e4.demo.mailapp");
    prefs.put("username", usernameField.getText());
    prefs.put("password", passwordField.getText());
    prefs.put("host", hostField.getText());

    try {
      prefs.flush();
      super.okPressed();
    } catch (BackingStoreException e) {
      ErrorDialog.openError(getShell(), "Error",
        "Error while storing preferences",
        new Status(IStatus.ERROR, "org.eclipse.e4.demo.mailapp", e.getMessage(),e)
      );
    }
  }
}
```

You notice that we are using DI here as well to get the current preference values. The useage of DI is not restricted to the framework but you can use it in your own code as well to create instances.

The implementation of the PreferenceHandler shows how one uses the DI-Framework in custom code to create an instance of a class.

```
@Execute
public void execute(IEclipseContext context) {
  PreferenceDialog dialog = ContextInjectionFactory.make(PreferenceDialog.class, context);
  dialog.open();
}
```



Connection informations

Configure the connection informations

Username: john

Password: doe

Host: tomsondev.bestsolution.at

Abbildung 21: Connection Configuration

The Event System

Another major change comeing with e4 is the event system provided. E4 does not invent its own but uses the EventAdmin-Bus provided by OSGi.

The first thing we do is to register ourselves as listener to the MailSession and posting an event into the system but before you'll have to add some more bundles:

- org.eclipse.e4.core.services
- org.eclipse.osgi.services

```
public class AccountView {

  //Unmodified

  private ISessionListener listener;

  @Inject
  @Optional
  private IEventBroker eventBroker;

  @Inject
  public AccountView(Composite parent, IMailSessionFactory mailSessionFactory) {

    //Unmodified
    listener = new ISessionListener() {
     public void mailAdded(IFolder folder, IMail mail) {
       if( eventBroker != null ) {
         Map<String,Object> map = new HashMap<String, Object>();
         map.put(EventConstants.NEW_MAIL_TAG_FOLDER, folder);
         map.put(EventConstants.NEW_MAIL_TAG_MAIL, mail);
         eventBroker.post(EventConstants.NEW_MAIL, map);
       }
     }
   };
  }

  //Unmodified

  @PostConstruct
  public void init() {
    if( username != null && password != null && host != null ) {
      if( mailSession != null ) {
        mailSession.removeListener(listener);
      }

      mailSession = mailSessionFactory.openSession(host, username, password);
      if( mailSession != null ) {
        viewer.setInput(mailSession.getAccounts());
        mailSession.addListener(listener);
      } else {
        viewer.setInput(new WritableList());
      }
    }
    modified = false;
  }
}
```

The code is quite straight forward the only interesting thing is that we are using **post** which means we are not blocking until all receivers priocessed the event. If we want the event to be deliver in a synchronous fashion we would have used **send**.

Code parts who want to get informed about those events are subscribing them to the EventBroker like this:

```
public class FolderView {
  private IFolder folder;

  @PostConstruct
  void hookEvents() {
    if( eventBroker != null ) {
      eventBroker.subscribe(EventConstants.NEW_MAIL, new EventHandler() {

        public void handleEvent(final Event event) {
          if( event.getProperty(EventConstants.NEW_MAIL_TAG_FOLDER) == folder ) {
            viewer.getControl().getDisplay().asyncExec(new Runnable() {
              public void run() {
                viewer.add(event.getProperty(EventConstants.NEW_MAIL_TAG_MAIL));
              }
            });
          }
        }
      });
    }
  }

  @Inject
  public void setFolder(@Named(IServiceConstants.ACTIVE_SELECTION) @Optional IFolder folder) {
    if( folder != null ) {
      this.folder = folder;
      viewer.setInput(folder.getSession().getMails(folder, 0, folder.getMailCount()));
    }
  }
}
```

It is important to mention that this is the ONLY event system available and all informations are passed through this system (e.g. creation of widgets like the shells for workbench windows, ...).

Contributing Fragments

Until now our application is created out of one single “monolithic” bundle and application model definition but that's not how a typical Eclipse-RCP-Application is made up. One of the strengths of Eclipse 3.x has been that an application could be made up from different bundles who contributed pieces to make up a complete application.

e4 is no different in this aspect but contributing to the model is done a bit differently. What you do is to contribute small fragments to a base application model and the e4-runtime incooperates those fragments into the final model used to create the application.