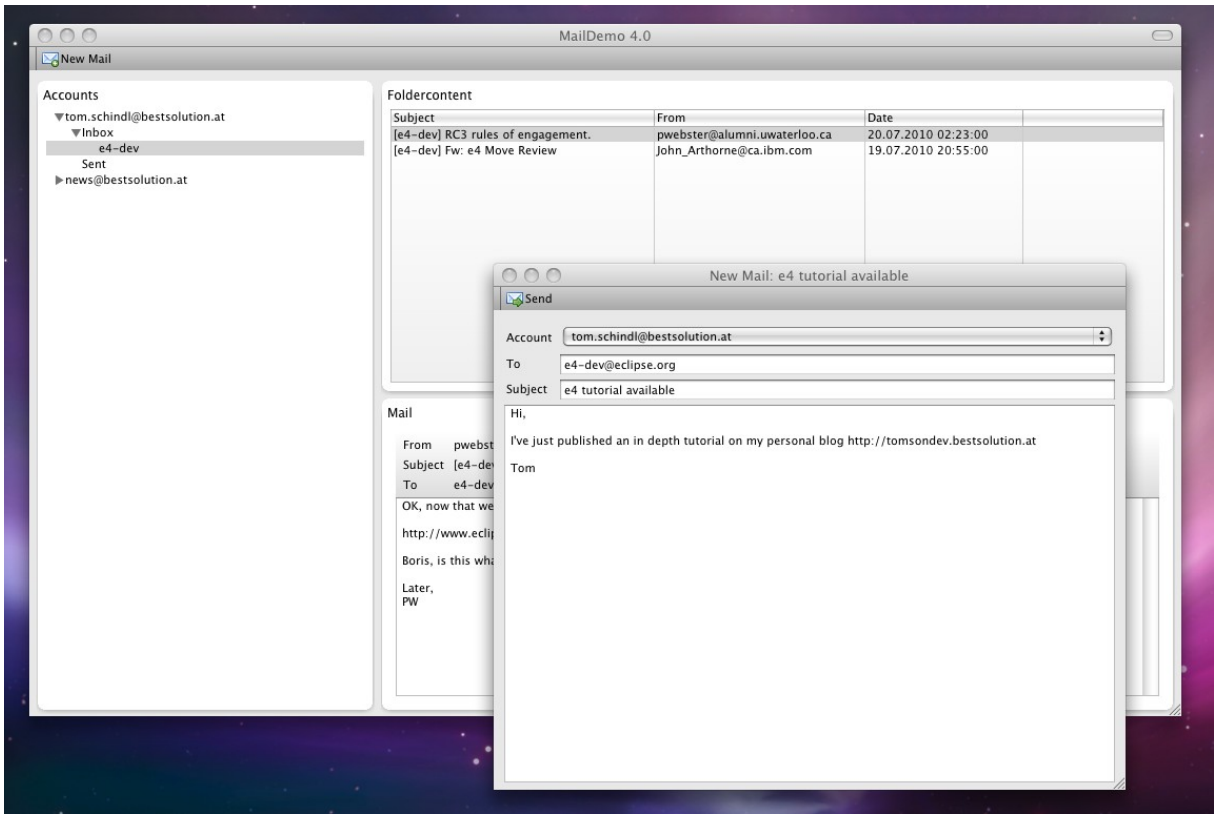
An Introduction to e4



How to write an RCP-Application with e4

Tom Schindl <tom.schindl@bestsolution.at>

Table of contents

Abstract

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 a similar application using technologies from e4.

This document will not delve into the details about about the internals of e4 but will instead focus on showing how one can use e4 to create an application.

We hope to deliver an in depth book talking about the internals and more advanced informations in the 4.1 timeframe.

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 a 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 your IDE we'd like to mention that it is not targeted yet 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.

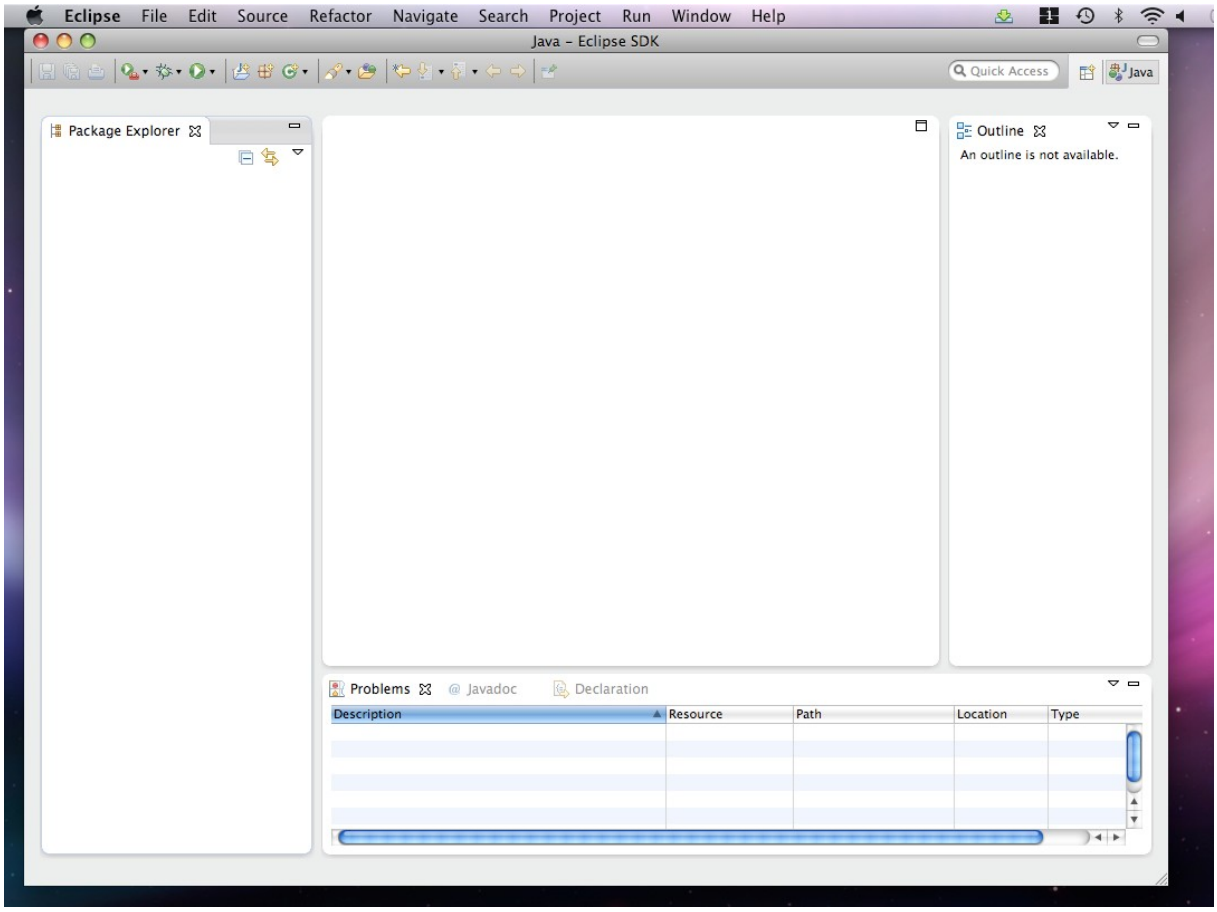


Figure 1: 4.0 SDK

Install e4 (Model)Tooling

The e4 tooling provides a specialized editor designed with the e4 model in mind and simplifies working with e4 application models.

If you don't want to use the tooling you can also use standard EMF-Tools to work with the application model but most people will probably prefer the specialized editor we are using through out this tutorial.

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

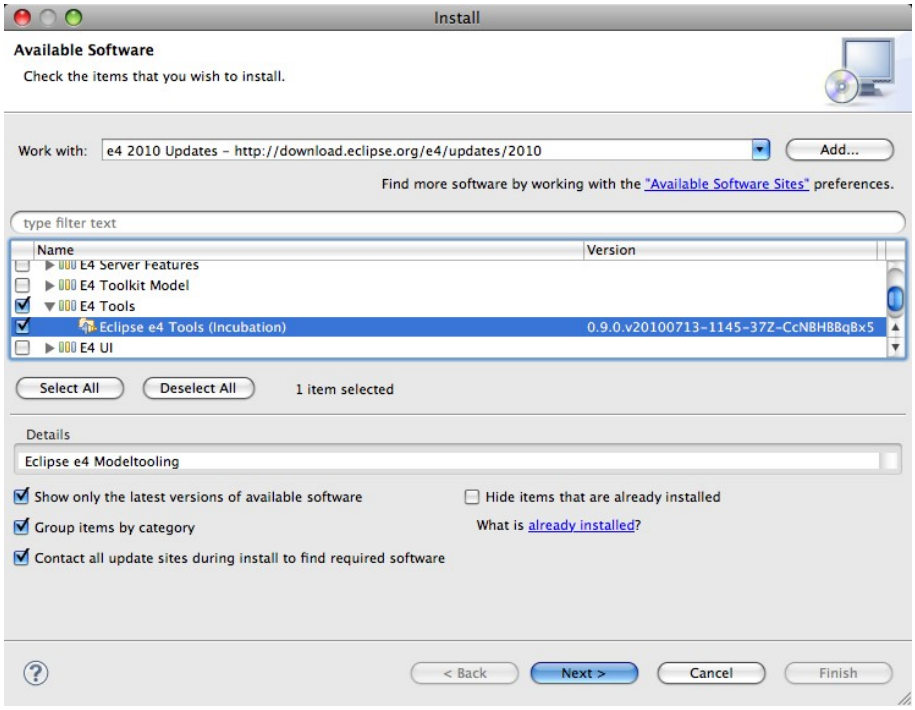


Figure 2: Install e4 Tooling

Setup project

There is a wizard for creating a complete e4 RCP Project but we will not be using that wizard. Instead, we will setup the project by hand so that we understand exactly what is going on in the back.

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:

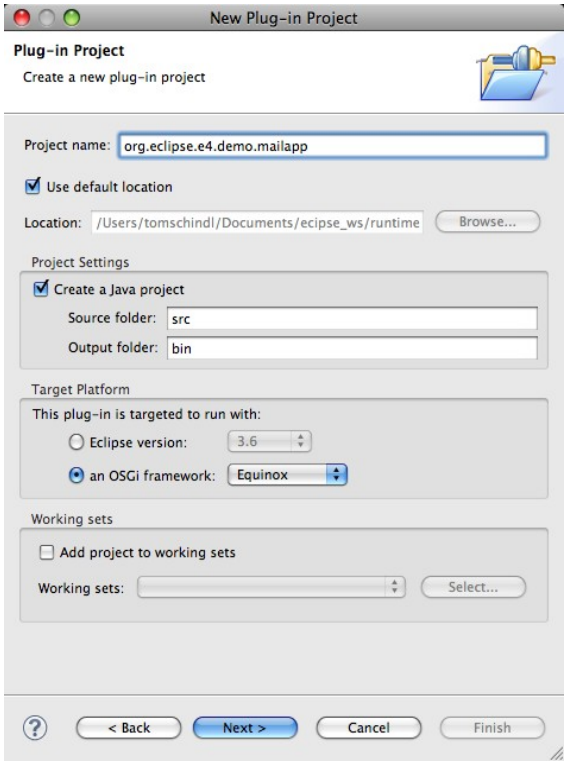


Figure 3: New OSGi-Project 1

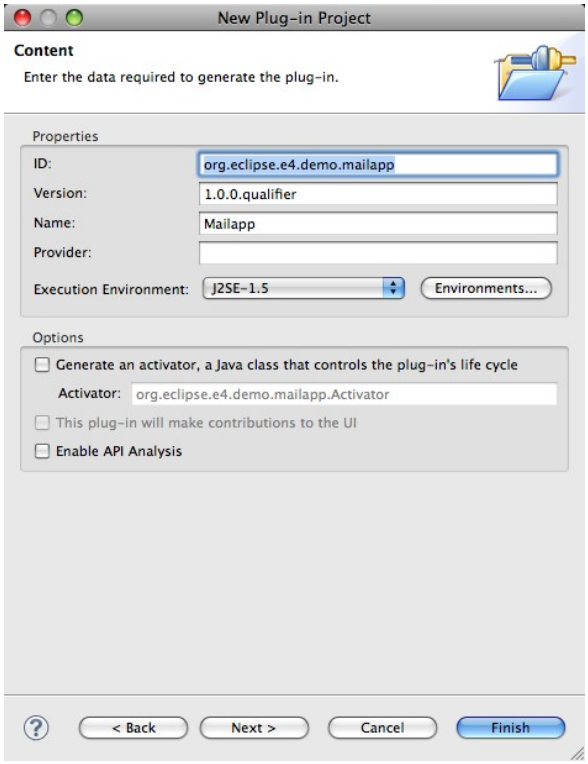


Figure 4: New OSGi-Project 2

Add a product definition

To create something we can launch and export we are creating an Equinox application and product definition by 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>
```

We are still missing two things before we can launch our application:

- An e4-Application which uses the predefined E4Application has to have a minimal workbench model (we'll learn about this in the upcoming sections)
- A .product to define a launchable and exportable 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:

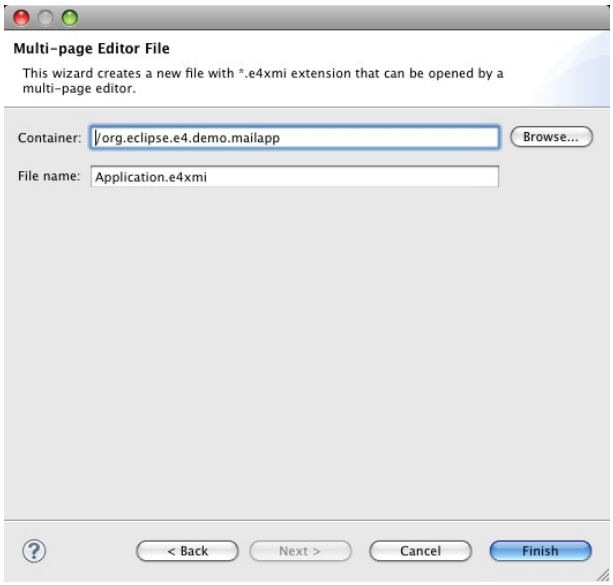


Figure 5: New Application Model

Technically this would be enough to launch an application but a 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:

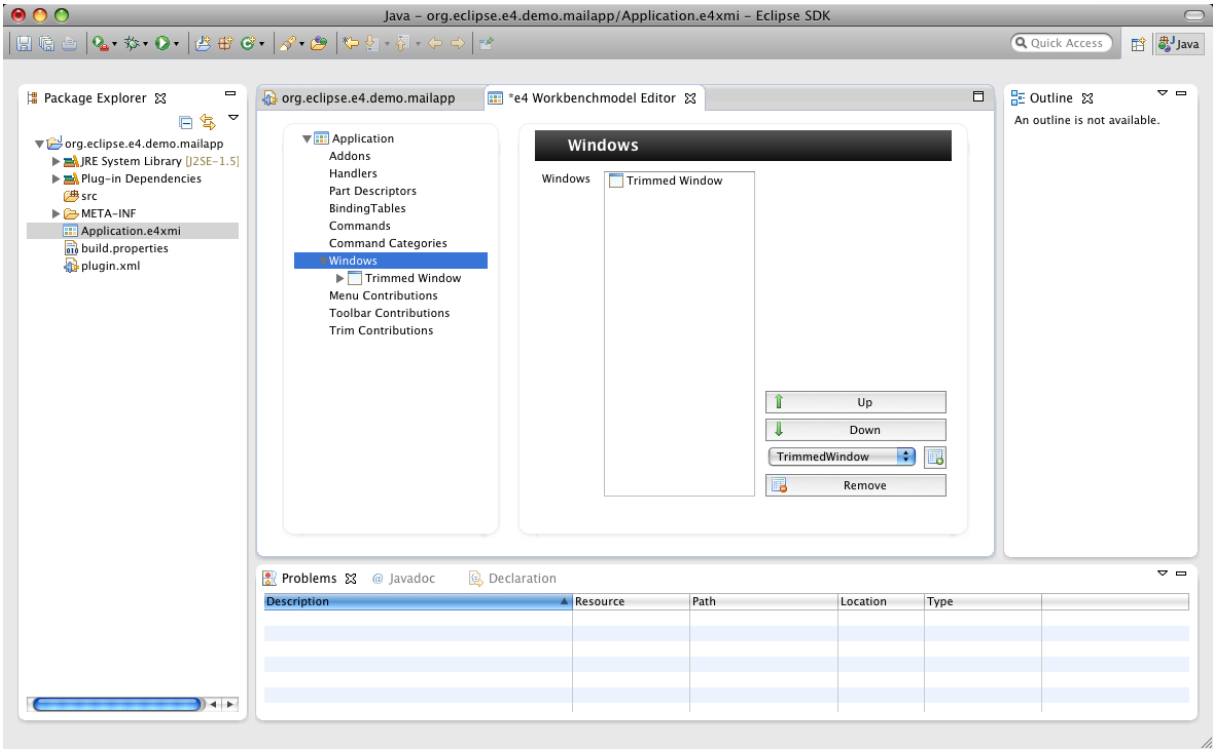


Figure 6: Add window to model

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. You should familiarize yourself 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 on its own.

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 event system it uses internally for communication and can be used by you as well
   - `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:

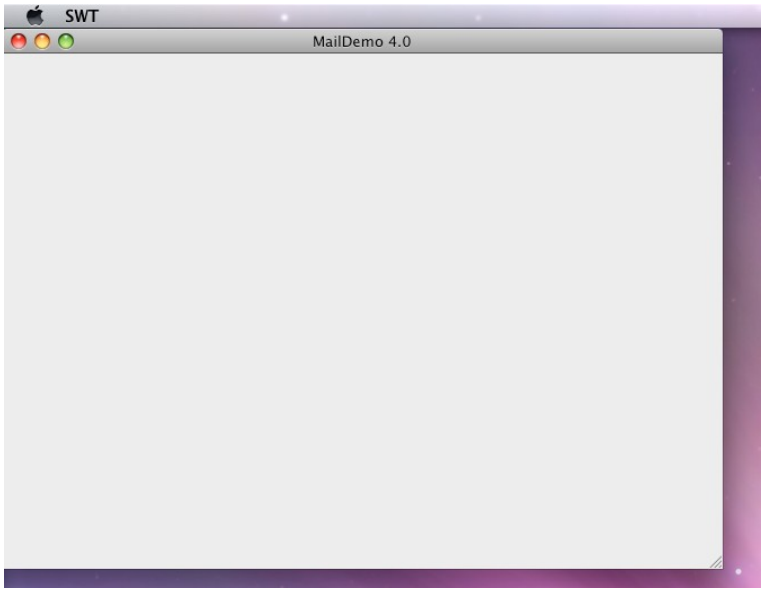


Figure 7: First Application

Want the source code? Look at "Source Zip for Chapter 2" on page 57.

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.mailservice.mock

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

The premade bundles are available from "Premade Service-Bundle" on page 57.

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:

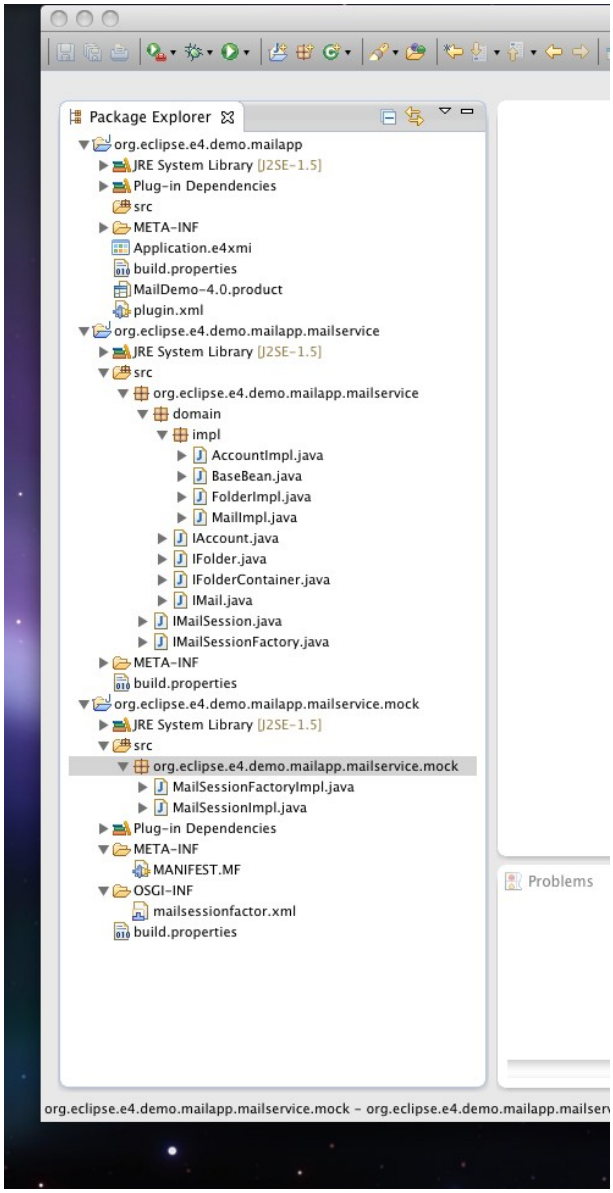


Figure 8: Workspace after ServiceBundles import

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());
    }
  }
}
```

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 classpath ourselves.

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:

```
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 and you should see something like this:

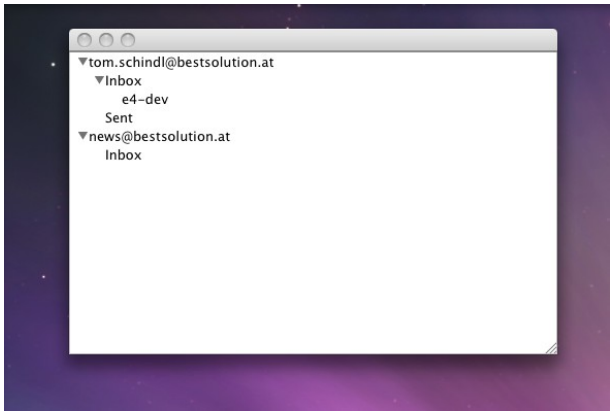


Figure 9: Test Account UI

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:

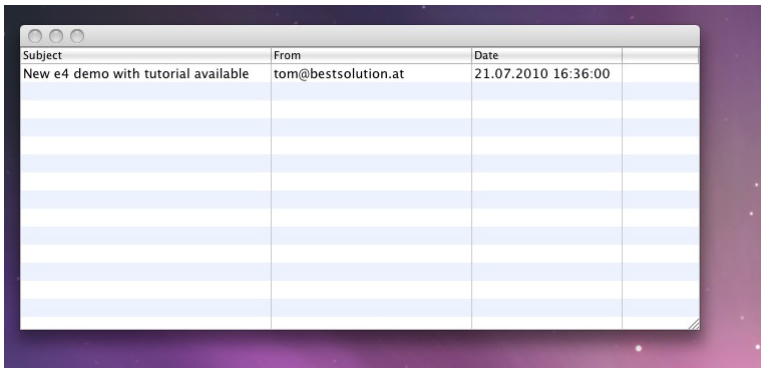


Figure 10: Test FolderView

Create the Mail UI

This UI displays a 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-Program should create a UI like this:

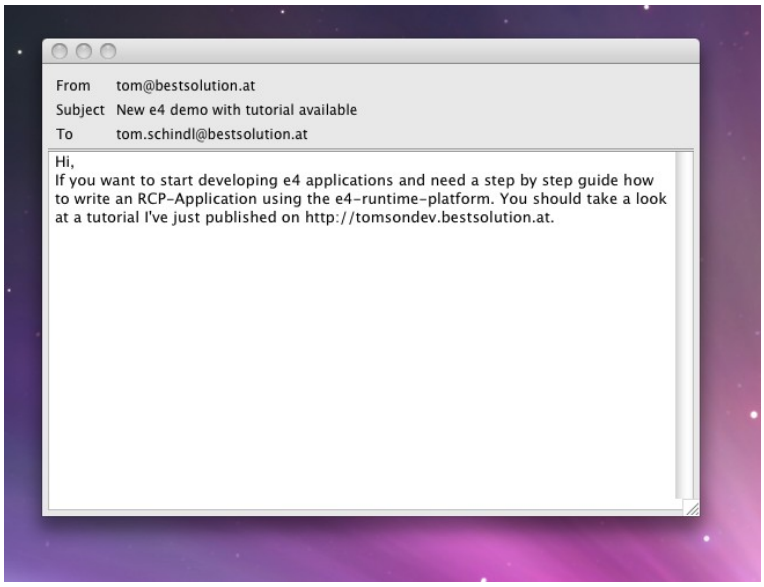


Figure 11: Test MailView

We have now created all the UI parts of our application without the need to have any knowledge about the e4-Framework. All we had to know was SWT/JFace and how to program in Java.

Want the source code? Look at "Source Zip for Chapter 4" on page 57.

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 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 few 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 it's not really not rocket science.

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
  void updateFolder(@Named(IServiceConstants.ACTIVE_SELECTION) 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 what is more interesting 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 interested in.

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 information we need to make up the UI.

There's no other way to get access to information 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 required to call the constructor are looked up in the so 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** usage 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 the required type the system should pass in null instead.

The other 2 annotations you see are controlling the life cycle of 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.

Wiring 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:

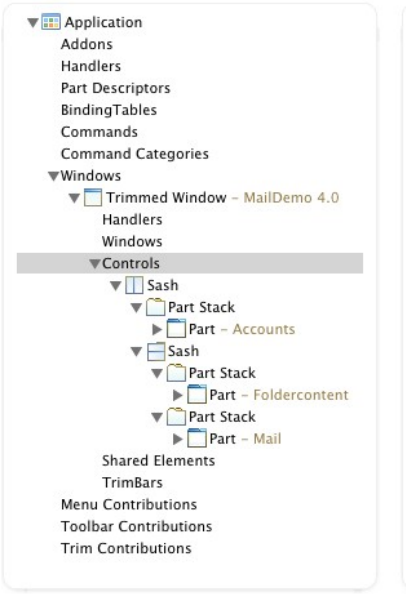


Figure 12: UI Model for Mail 4.0

b) Select the 1st Part in the tree and press the "Find..."-button on the Class URI attribute and search for our `AccountView`-POJO.

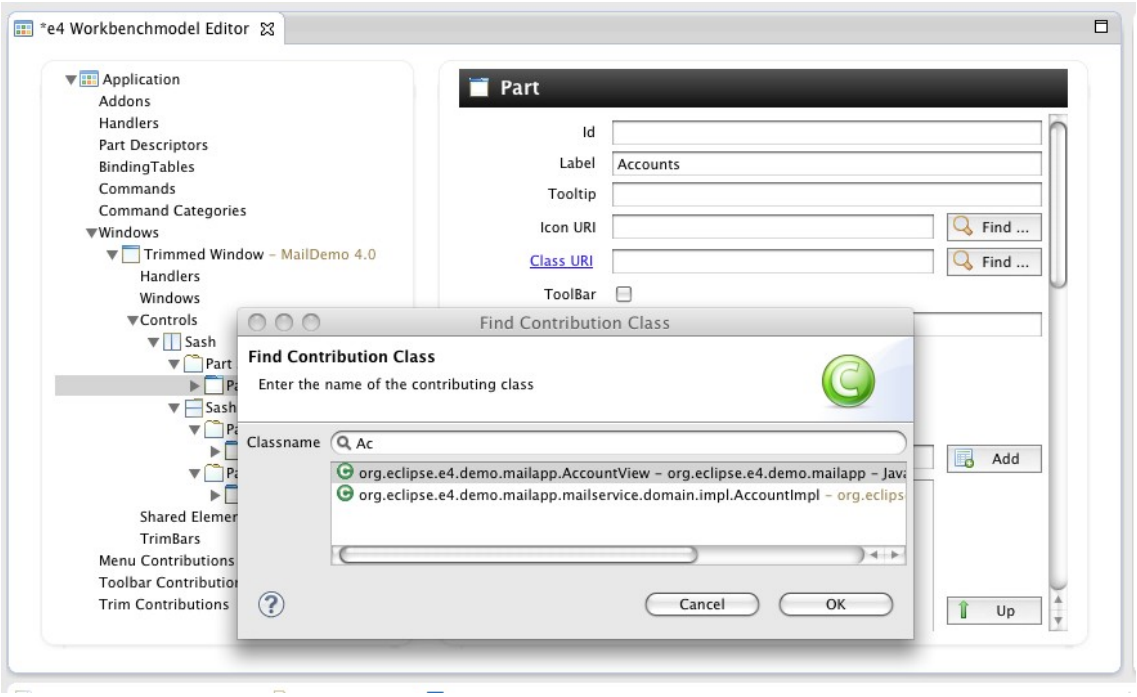


Abbildung 13: Connect POJO with UI

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 steps 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:

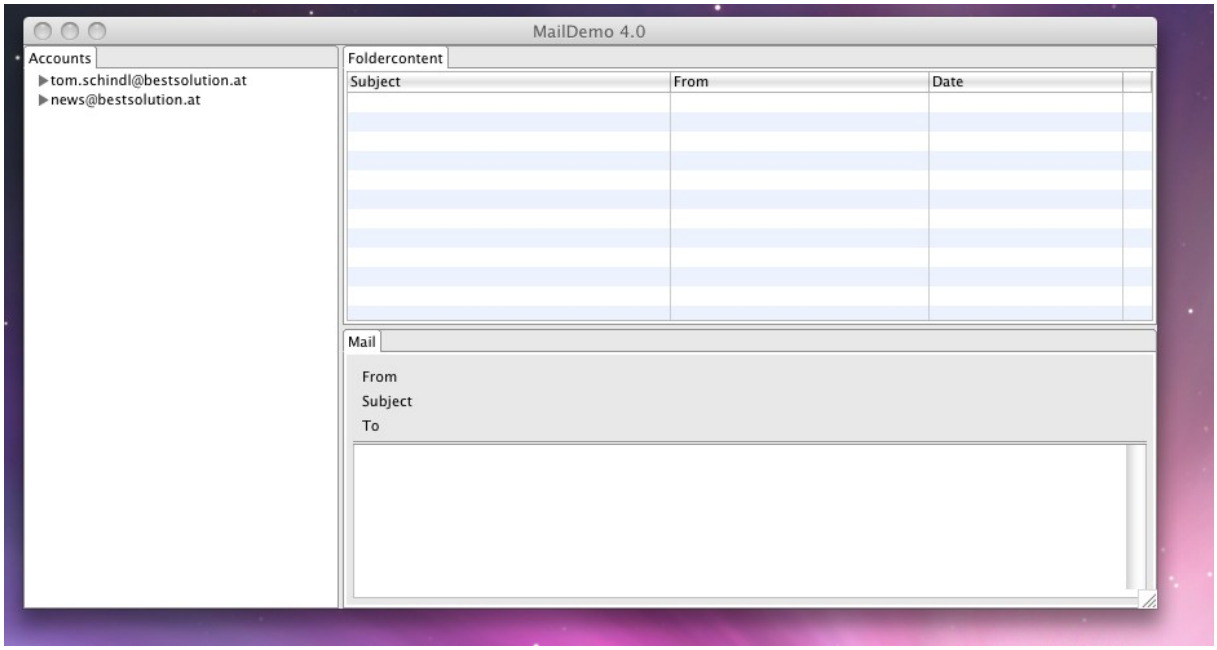


Figure 14: Running MailDemo Application

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 a 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.

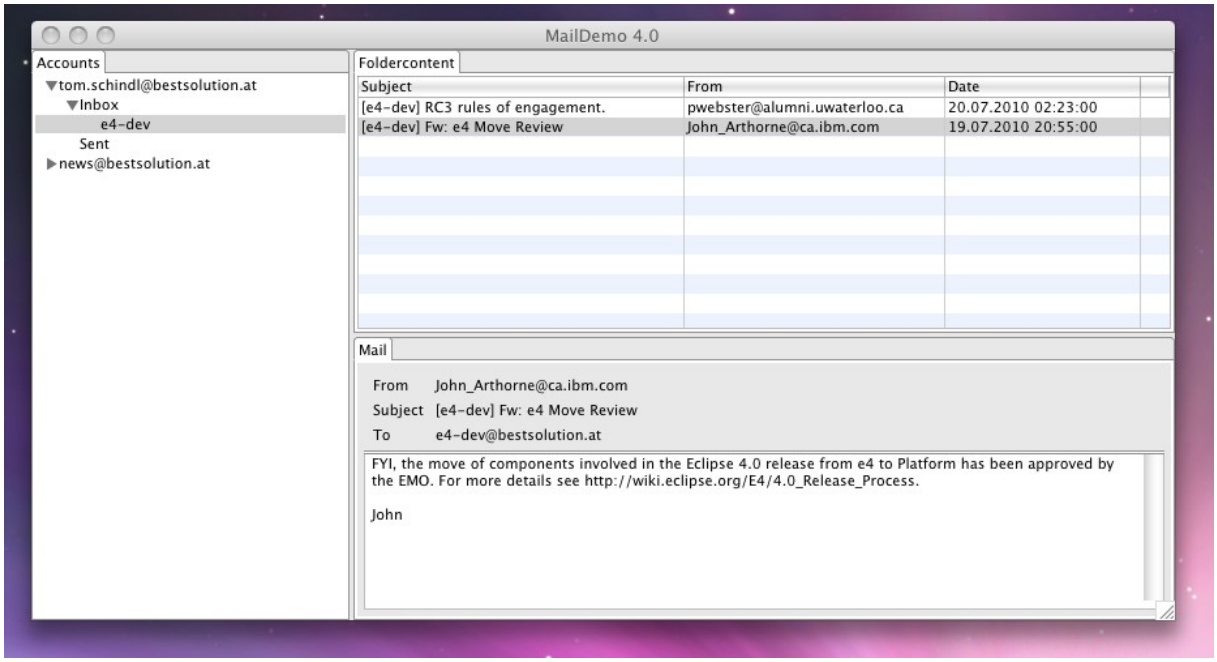


Figure 15: Running MailDemo Application after chapter 5

Want the source code? Look at "Source Zip for Chapter 5" on page 57.

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 a CSS-like declarative syntax. To inform the framework about the CSS-stylesheet it has to 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 to the whole RCP-Application.

Defining the above property is all you need to do to tell your application to consume a CSS stylesheet.

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.renderers.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 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.

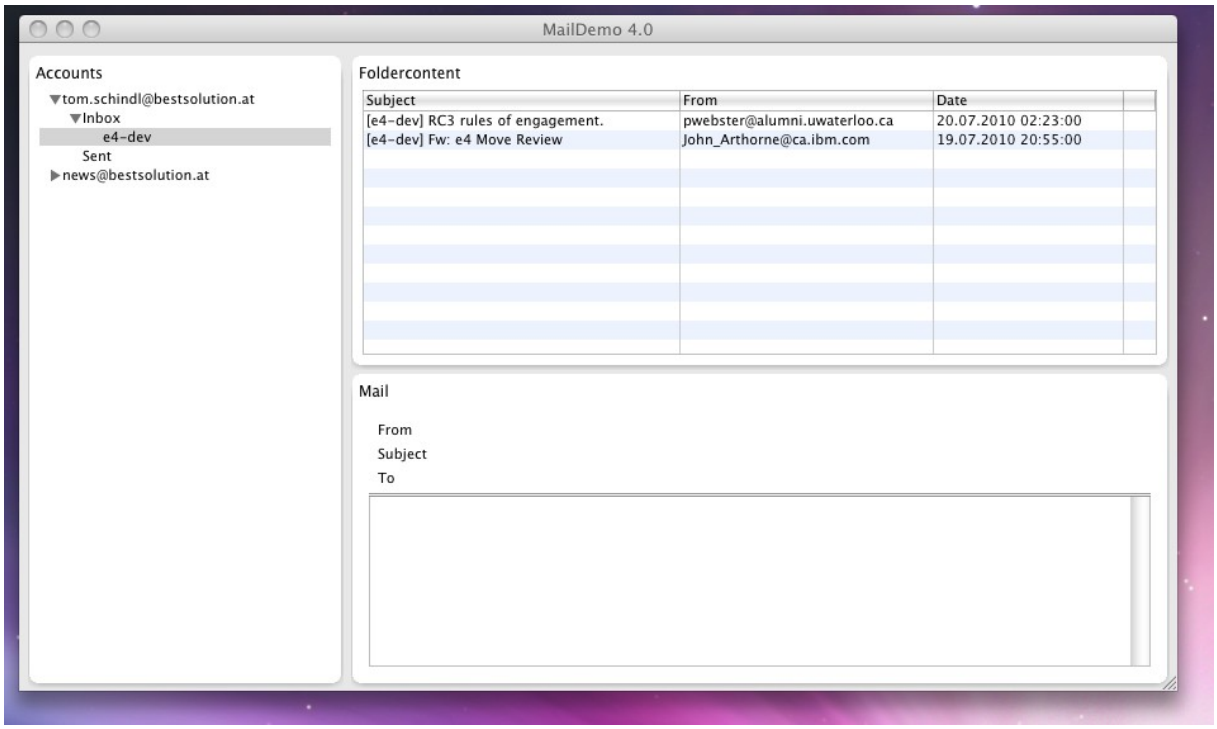


Figure 16: MailDemo with CSS

The first set of style informations have been applied to widget owned by the e4-application framework but we'd also like to apply css informations on the widgets created by the Part-POJOs.

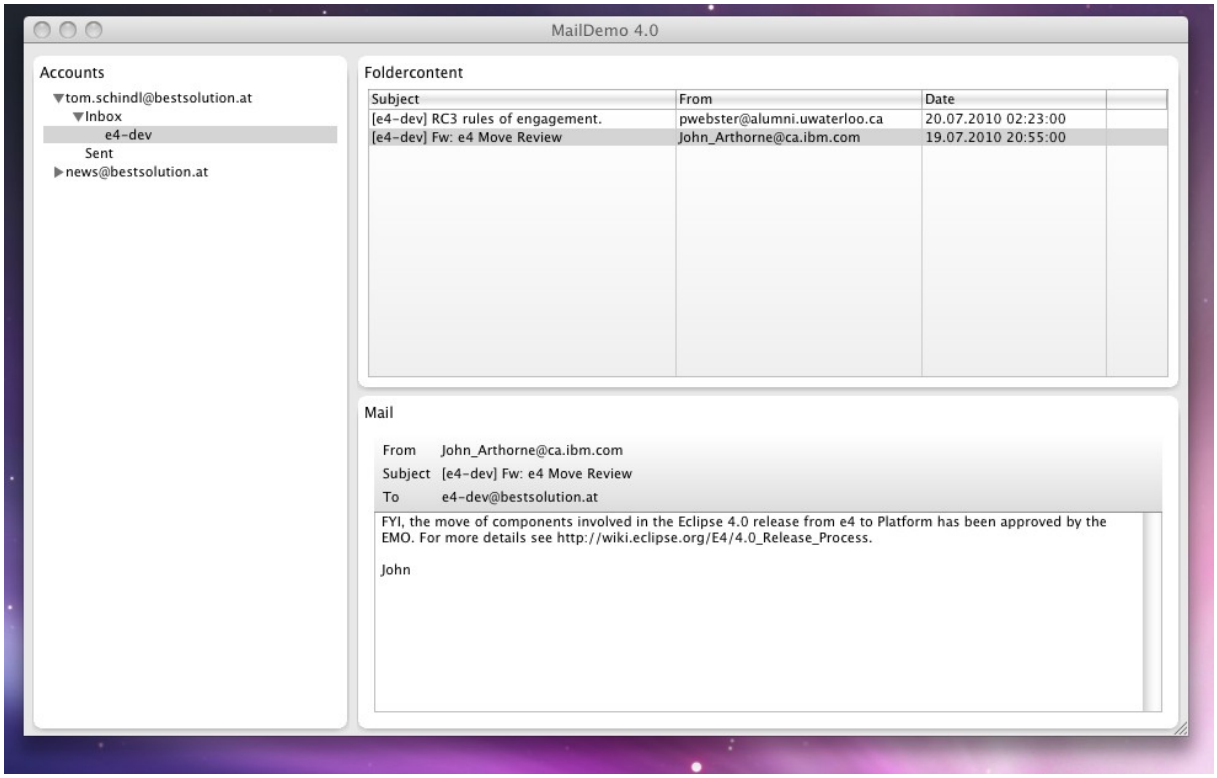


Figure 17: CSS in custom area

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 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 CSS-class on the widget.

Want the source code? Look at "Source Zip for Chapter 6" on page 57.

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;
    this.modified = true;
  }

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

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

  @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.");
      }
    }
  }
}
```

Want the source code? Look at "Source Zip for Chapter 7" on page 57.

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.

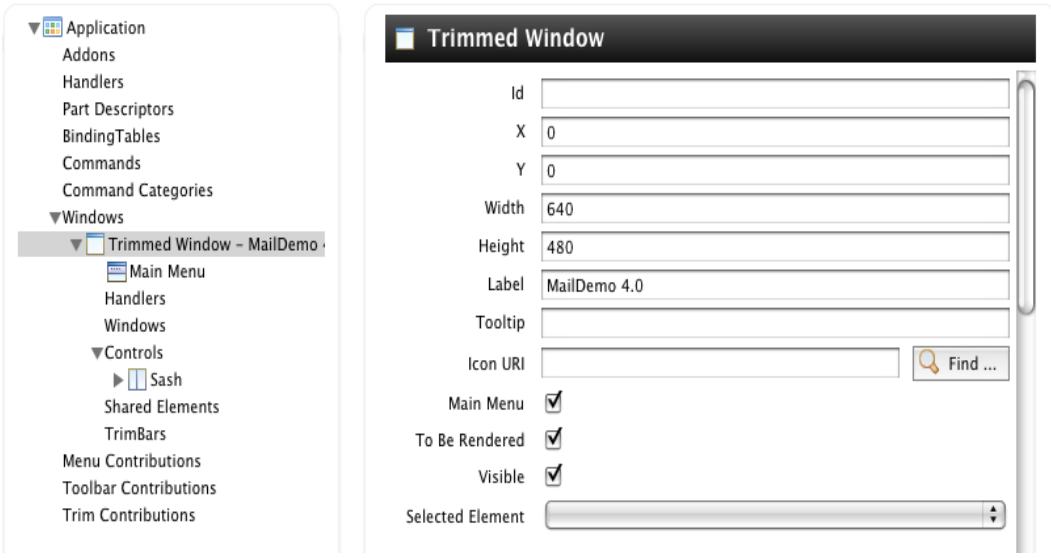


Figure 18: Add main menu

c) Create a structure like this

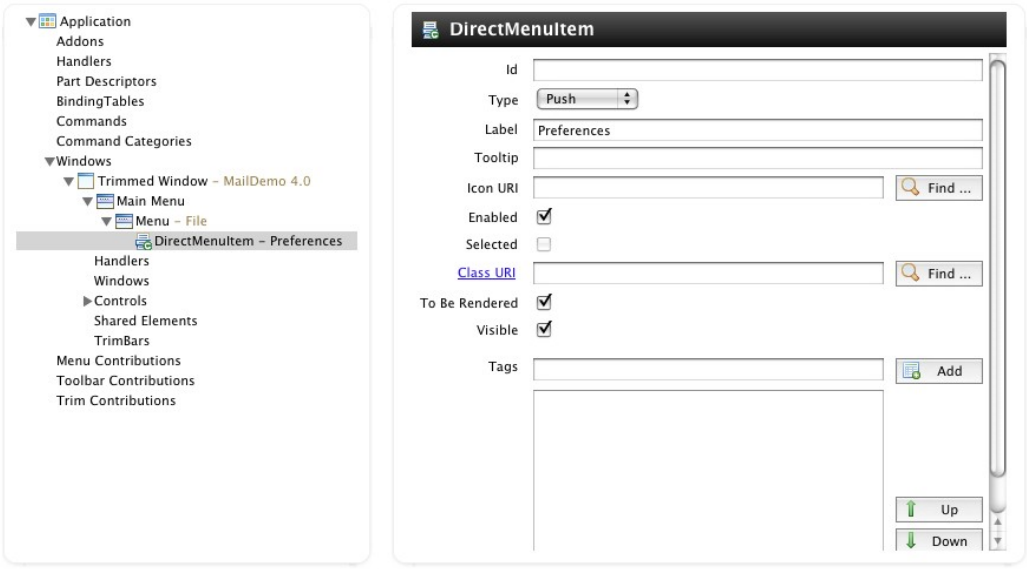


Figure 19: Create the menu structure

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

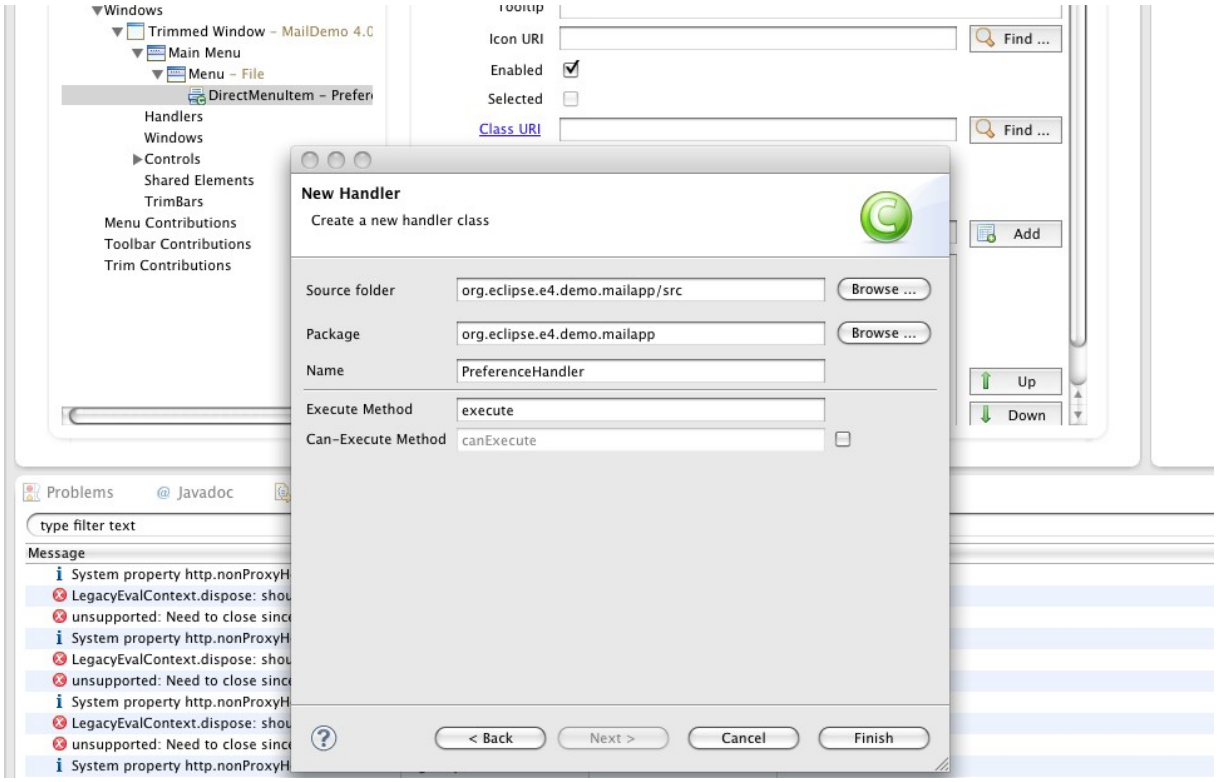


Figure 20: Create handler class

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();
}
```

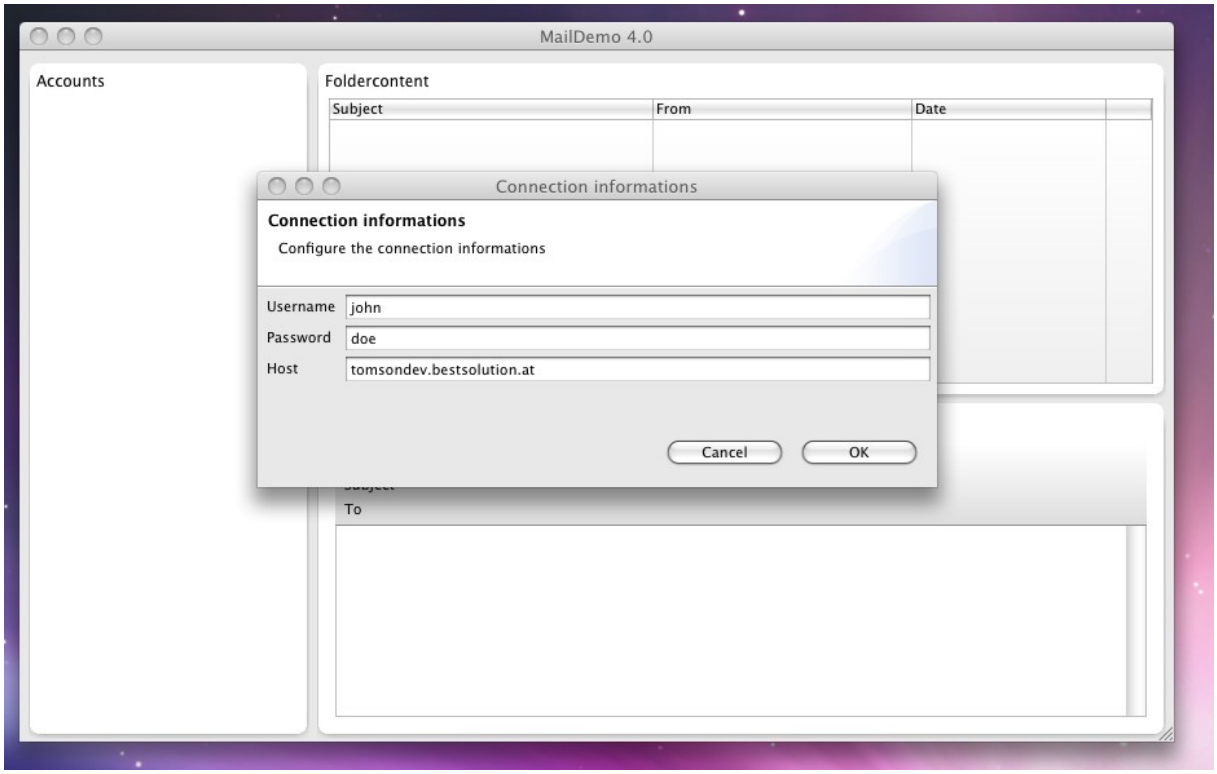


Figure 21: Connection Configuration

Want the source code? Look at "Source Zip for Chapter 8" on page 57.

The Event System

Another major change coming 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 a listener to the `MailSession` and **`post`** an event into the system. Before writing the code 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;
  }

  @PreDestroy
  void cleanUp() {
    if( mailSession != null && listener != null ) {
      mailSession.removeListener(listener);
    }
  }
}
```

The code is quite straightforward, the only interesting thing is that we are using **post** which means we are not blocking until all receivers processed 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 {

  //Unmodified

  @Inject
  @Optional
  private IEventBroker eventBroker;
  private EventHandler eventHandler;

  private IFolder folder;

  //Unmodified
```

```
  @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));
              }
            });
          }
        }
      });
    }
  }

  @PreDestroy
  void unhookEvents() {
    if( eventBroker != null && eventHandler != null ) {
      eventBroker.unsubscribe(eventHandler);
    }

  }

  @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 note that this is the ONLY event system available and all notifications are passed through this system (e.g. creation of widgets like the shells for workbench windows, ...).

Want the source code? Look at "Source Zip for EventSystem" on page 57.

Advanced e4 - 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 merges those fragments into the final model used to create the application.

A first and very important thing when contributing is that the elements in the

base model we want to contribute to have to have an `elementId` (in the editor shown as "Id").

You should open our `Application.e4xmi` and check:

- the Application-Element has the Id: `org.eclipse.e4.demo.mailapp.application`
- the File-Menu-Element has the Id: `org.eclipse.e4.demo.mailapp.filemenu`

The next thing we do is to create new OSGi-bundle named "`org.eclipse.e4.demo.mailapp.newmail`" and add the following dependencies:

- `org.eclipse.e4.core.di`
- `javax.inject`
- `org.eclipse.e4.ui.services`
- `org.eclipse.swt`
- `org.eclipse.e4.demo.mailapp.mailservice`
- `org.eclipse.e4.ui.model.workbench`
- `org.eclipse.e4.ui.workbench`
- `org.eclipse.e4.core.contexts`
- `javax.annotation`
- `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`

Now we create a Model-Fragment using "File > New > Other ..." and "e4 > Model > New Model Fragment".

We are going to contribute the following things to the base model:

- A Command to the Application-element
- A Handler to the Application-element
- A HandledMenuItem to the File-Menu-element

Contributing a Command-Element

The first element we contribute is a Command-Element which gets contributed to the Application-Element of our base model.

a) In the Fragment-Editor we select the ModelFragments entry and press the "Add ..."-button

b) We press the "Find ..."-button next to the "Element Id"-TextField which helps us to find the correct element-id of the application-element

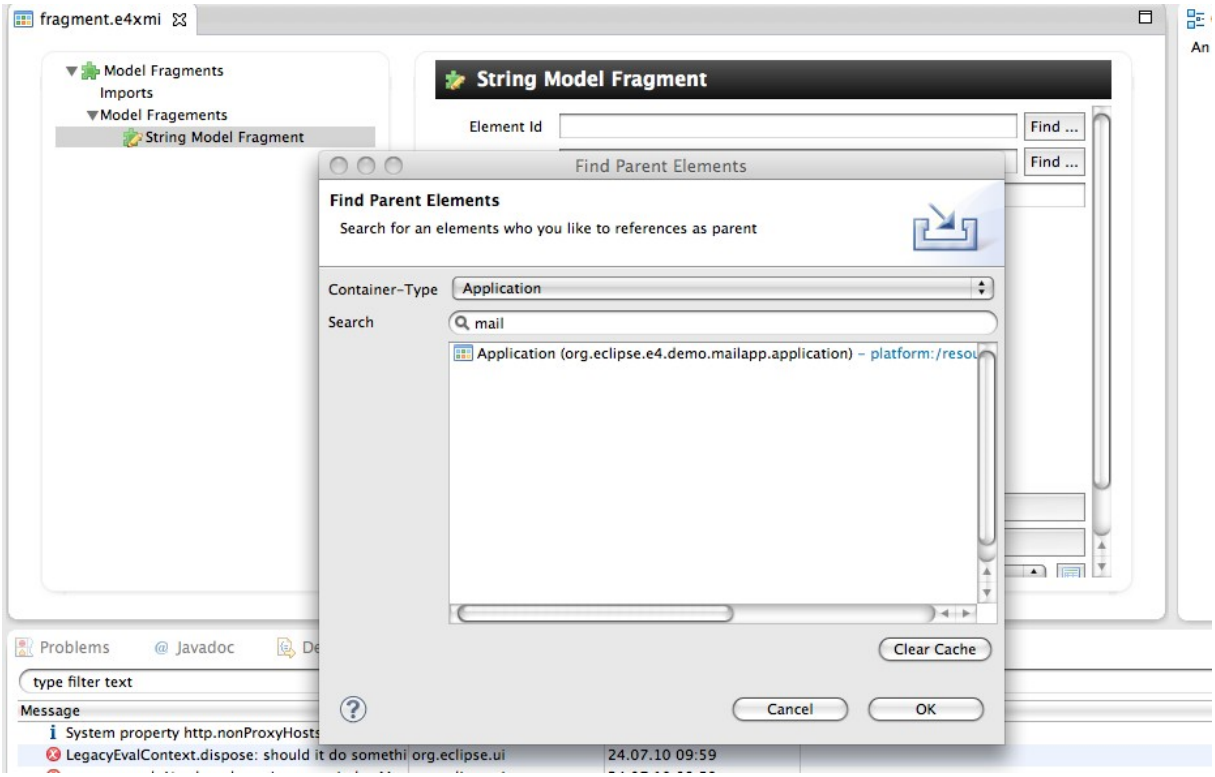


Figure 22: Select Application-Element Id

c) We press the "Find ..."-button next to the "Featurename"-TextField which helps us to find the feature-name (=attribute-name) the contributed Command should be stored in.

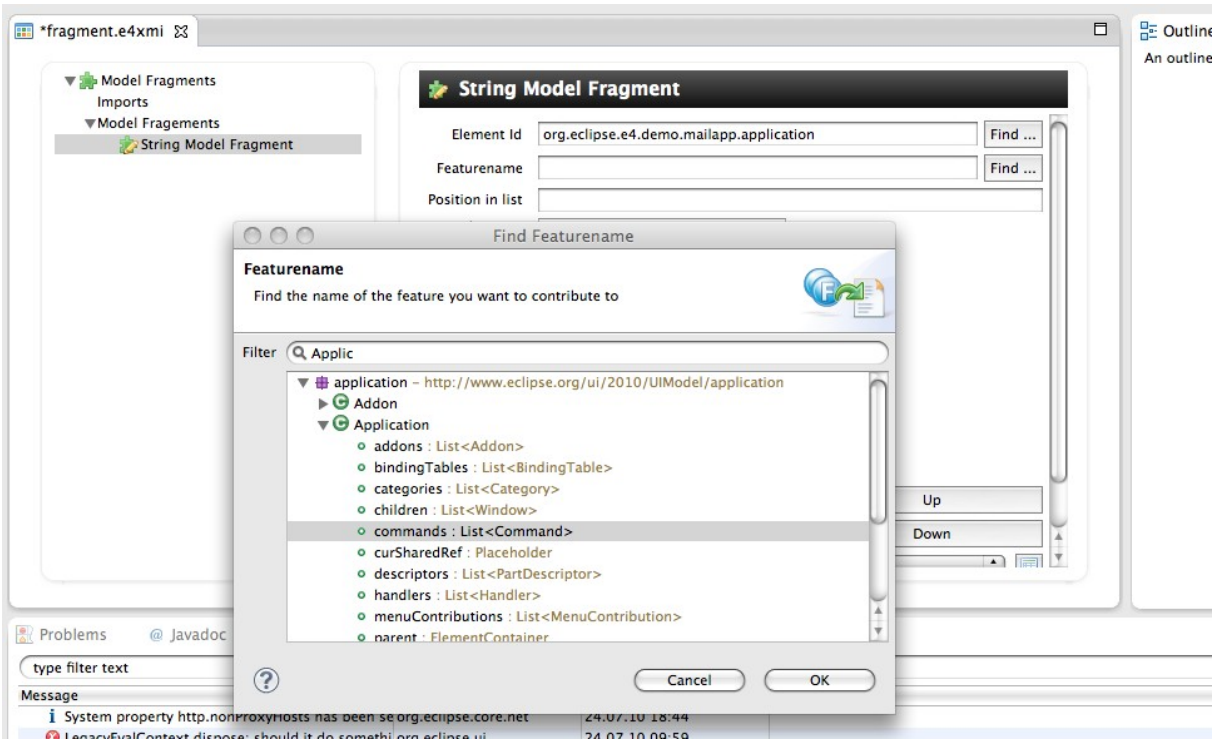


Figure 23: Select Featurename

d) Select Command in the drop down click the button next to it

e) Fill the "Element Id" and "Name"-Fields like this:

Figure 24: Edit Command Element

Contributing a Handler

The next element we contribute is a Handler which gets connected to our command and and will be called when the command is executed.

a) Add another “String Model Fragment”

b) Set “Element Id” to “org.eclipse.e4.demo.mailapp.application” and “Featurename” to “handlers”

c) Add a Handler-Element

d) Press the “Find...”-button next to the Command text field

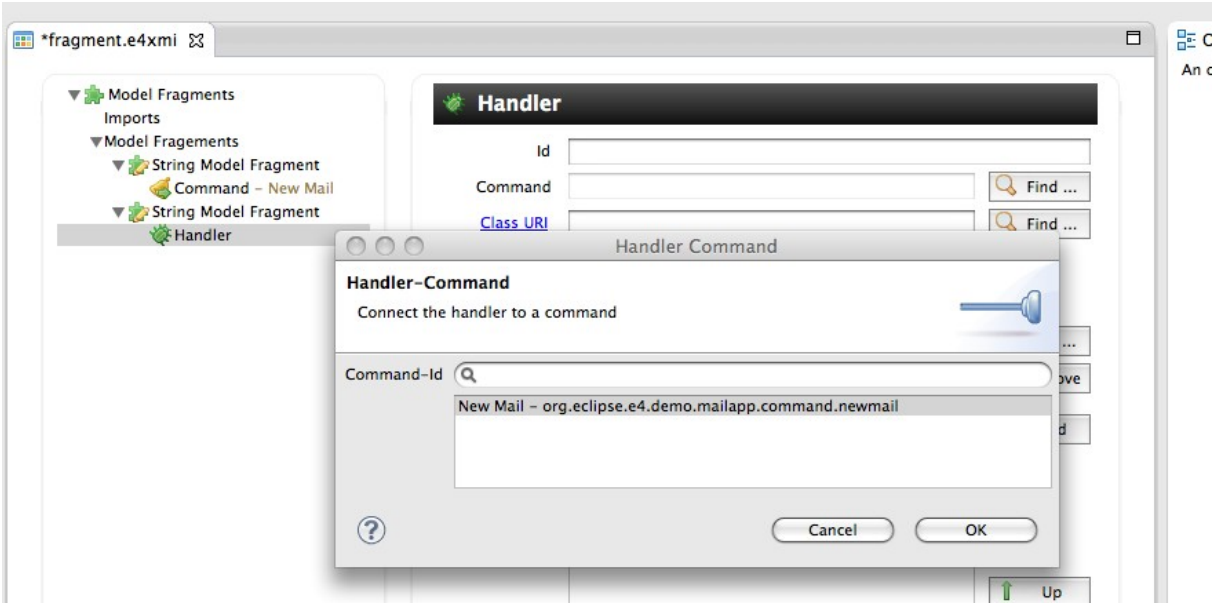


Figure 25: Connect to command

e) Press the “Class URI”-Link and enter the following information

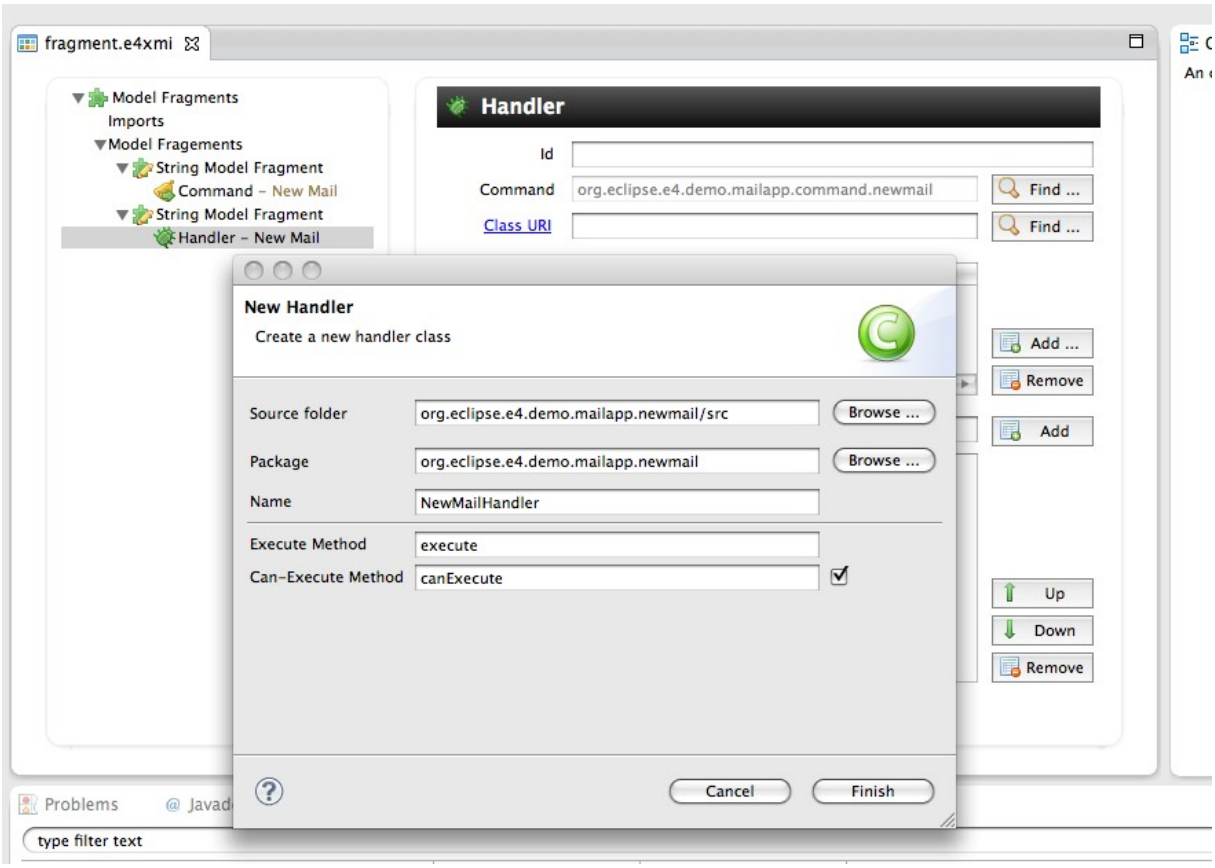


Figure 26: Set a handler class

Contributing a HandledMenuItem

The next thing we contribute is MenuItem to the File-Menu which is connected to our command.

a) Add another "String Model Fragment"

b) Set "Element Id" to "`org.eclipse.e4.demo.mailapp.filemenu`", "Featurename" to "`children`" and "Position in list" to "`first`"

c) Add an HandledMenuItem-Element

d) Set the "Label"-Attribute to "New Mail"

e) Press the "Find ..."-button next to the "Command"-TextField

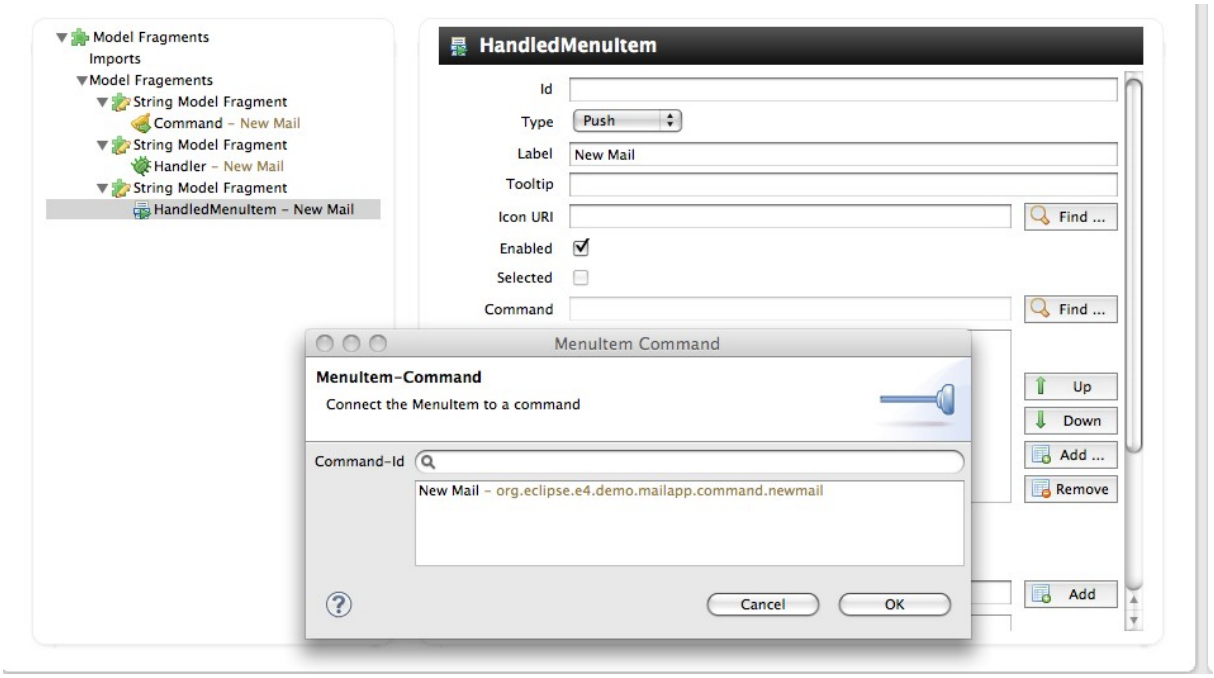


Figure 27: Connect MenuItem to Command

Contributing a ToolItem

Open the `Application.e4xmi` and add

- a `TrimBar` with side-value set to `Top`
- a `ToolBar` with `Id` set to "`org.eclipse.e4.demo.mailapp.maintoolbar`"

The model should now look like this:

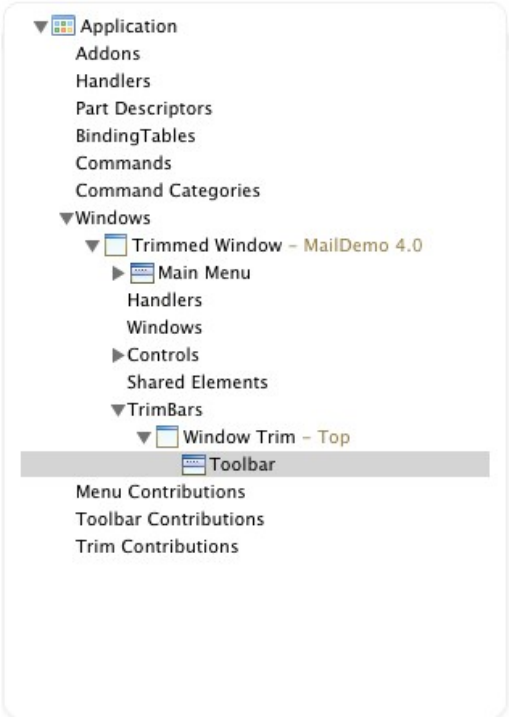


Figure 28: Toolbar Addition

Open the default.css-File and add the following entry:

```
.MTrimBar {
    background-color: #CFCFCF #A8A8A8 100%;
}
```

Back to our `fragement.e4xmi` execute the following steps:

a) Add another "String Model Fragment"

b) Set "Element Id" to "`org.eclipse.e4.demo.mailapp.maintoolbar`" and "Featurename" to "`children`"

c) Add a HandledToolItem, set its label to "New Mail" and connect it also to the "`org.eclipse.e4.demo.mailapp.command.newmail`"-command

We have to add 2 more elements:

- Command-Element and the following properties:
  - Id: org.eclipse.e4.demo.mailapp.command.sendmail
  - Name: Send Mail
- Handler:
  - connected to org.eclipse.e4.demo.mailapp.command.sendmail-command

- Class-URI: pointing to a class named SendMail

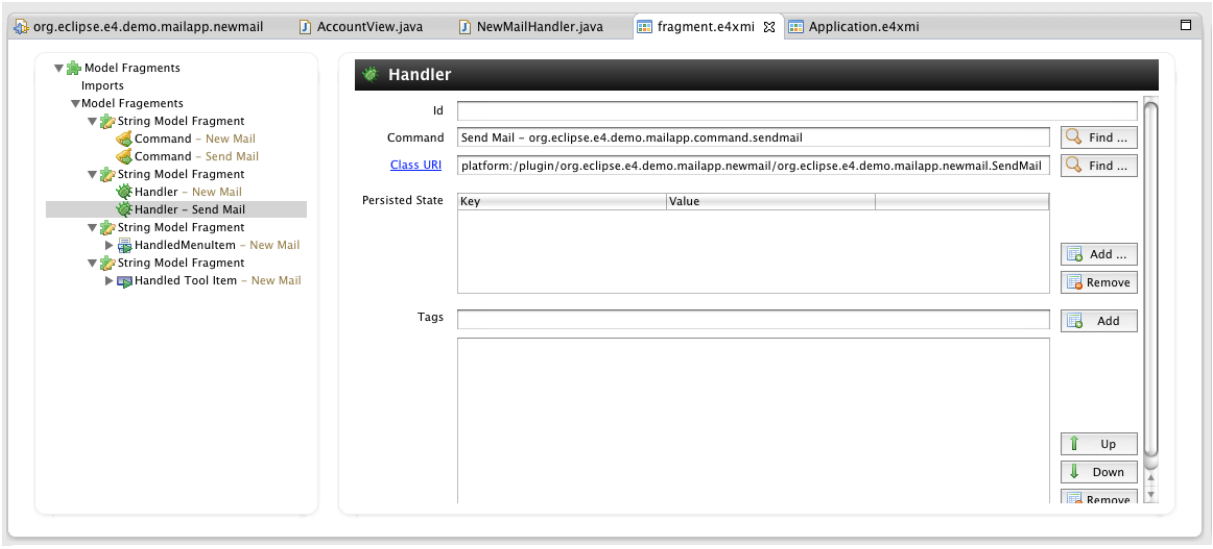


Figure 29: Final Fragment

Dynamic UI creation

That's it for the model. Let's now work on the Java code of the handler.

```
public class NewMailHandler {

  @Execute
  public void execute(final MApplication application, IMailSession mailSession) {
    // Create the window
    final MTrimmedWindow window = MBasicFactory.INSTANCE.createTrimmedWindow();
    window.getTags().add("temporaryObject");
    window.setHeight(500);
    window.setWidth(600);

    // Create the toolbar
    MTrimBar topTrim = MBasicFactory.INSTANCE.createTrimBar();
    topTrim.setSide(SideValue.TOP);
    window.getTrimBars().add(topTrim);

    MToolBar toolbar = MMenuFactory.INSTANCE.createToolBar();
    topTrim.getChildren().add(toolbar);

    MHandledToolItem sendItem = MMenuFactory.INSTANCE.createHandledToolItem();
    sendItem.setLabel("Send");
    sendItem.setIconURI("platform:/plugin/org.eclipse.e4.demo.mailapp.newmail/images/email_go.png")
;

    for( MCommand cmd : application.getCommands() ) {
      if( "org.eclipse.e4.demo.mailapp.command.sendmail".equals(cmd.getElementId()) ) {
        sendItem.setCommand(cmd);
      }
    }

    toolbar.getChildren().add(sendItem);

    // Create the mail editor
```

```
    MPart part = MBasicFactory.INSTANCE.createPart();
    part.setContributionURI(
      "platform:/plugin/org.eclipse.e4.demo.mailapp.newmail/org.eclipse.e4.demo.mailapp.newmail.EditMailView");
    window.getChildren().add(part);
    application.getChildren().add(window);
    window.getContext().set(IMail.class, mailSession.createMail());
  }

  @CanExecute
  public boolean canExecute(@Optional IMailSession mailSession) {
    return mailSession != null;
  }
}
```

What we are doing here is that we are programmatically interfacing with the application model to add a new window which has a toolbar at the top and a part to edit a mail that will take up the rest of the window's area.

I think what you see above is one of the coolest things in e4 comes with. You have access to the workbench model and are allowed to modify it at runtime, adding, removing, moving things around.

We also introduced a new `@CanExecute`-annotation we can use on Command-Handlers to compute the enabled state. The framework calls this method whenever the values defined in the parameters of the method changes, in our case the new command is only enabled if there's a `IMailSession` available.

Before we can run the application we need to make the some changes to our existing code.

```
public class AccountView {
  // Unmodified code

  @Inject
  @Optional
  private MApplication application;

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

    // Work around for 4.0 Bug of not cleaning up on Window-close
    viewer.getControl().addDisposeListener(new DisposeListener() {
      public void widgetDisposed(DisposeEvent e) {
        cleanUp();
      }
    });
  }

  // Unmodified code

  @PostConstruct
  public void init() {
    // Unmodified code

    if( application != null ) {
      application.getContext().set(IMailSession.class, mailSession);
    }
```

```
  }

  // Unmodified code

  @Focus
  void onFocus(@Named(IServiceConstants.ACTIVE_SHELL) Shell shell) {
    // Unmodified code

    if( application != null ) {
      application.getContext().set(IMailSession.class, mailSession);
    }
  }
}
```

What we do is that we get access to the MApplication-Element to get easy access to the context attached to it and store the current mail-session there. You see that I also introduced a work-around for a bug in the 4.0-code base which deals with **@PreDestroy** when a window is closed. This bug hasn't been a problem until because we only had a single window but is going to affect us now.

We also have to fix the problem in the FolderView:

```
public class FolderView {
  // Unmodified code

  @Inject
  public FolderView(Composite parent, @Optional IStylingEngine styleEngine) {
    // Unmodified code

    // Work around for 4.0 Bug of not cleaning up on Window-close
    viewer.getControl().addDisposeListener(new DisposeListener() {
      public void widgetDisposed(DisposeEvent e) {
        unhookEvents();
      }
    });
  }
}
```

The last 2 things missing now are:

a) The implementation of our EditMailView:

```
public class EditMailView {
  @Inject
  private MTrimmedWindow window;

  private ComboViewer viewer;

  private IObservableValue master = new WritableValue();

  private ObservablesManager manager = new ObservablesManager();

  @Inject
  public EditMailView(final Composite container) {
    manager.runAndCollect(new Runnable() {
      public void run() {
        createUI(container);
      }
    });
```

```
}

private void createUI(Composite container) {
  Composite parent = new Composite(container,SWT.NONE);
  parent.setLayout(new GridLayout(2, false));

  Label l = new Label(parent, SWT.NONE);
  l.setText("Account");

  viewer = new  ComboViewer(parent);
  viewer.getControl().setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
  viewer.setLabelProvider(new LabelProvider() {
    @Override
    public String getText(Object element) {
      return ((IAccount)element).getName();
    }
  });
  viewer.addSelectionChangedListener(new ISelectionChangedListener() {
    public void selectionChanged(SelectionChangedEvent event) {
      window.getContext().set(IAccount.class,
        (IAccount)((IStructuredSelection)event.getSelection()).getFirstElement());
    }
  });
  viewer.setContentProvider(new ObservableListContentProvider());

  DataBindingContext dbc = new DataBindingContext();
  IWidgetValueProperty prop = WidgetProperties.text(SWT.Modify);

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

  Text t = new Text(parent, SWT.BORDER);
  t.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
  dbc.bindValue(prop.observe(t), BeanProperties.value("to").observeDetail(master));

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

  t = new Text(parent, SWT.BORDER);
  t.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
  dbc.bindValue(prop.observe(t), BeanProperties.value("subject").observeDetail(master));

  BeanProperties.value("subject").observeDetail(master).addValueChangeListener(
    new IValueChangeListener() {
      public void handleValueChange(ValueChangeEvent event) {
        String value = (String) event.diff.getNewValue();
        value = value == null ? "" : value;

        if( window != null ) {
          window.setLabel("New Mail: " + value);
        }
      }
  });


  t = new Text(parent, SWT.BORDER|SWT.MULTI|SWT.WRAP);
  GridData gd = new GridData(GridData.FILL_BOTH);
  gd.horizontalSpan = 2;
  t.setLayoutData(gd);
  dbc.bindValue(prop.observe(t), BeanProperties.value("body").observeDetail(master));
}
```

```
  @PreDestroy
  void cleanUp() {
    manager.dispose();
  }

  @Inject
  void setMailSession(IMailSession session) {
    if( session != null ) {
      viewer.setInput(session.getAccounts());
    } else {
      if( ! viewer.getControl().isDisposed() ) {
         viewer.setInput(new WritableList());
      }
    }
  }

  @Inject
  @Optional
  void setMail(IMail mail) {
    master.setValue(mail);
  }
}
```

Not really a lot of magic beside that we are here pushing the ComboViewers-Account selection into the TrimmedWindow's context.

When we introduced DI I said you can think of the `IEclipseContext` used for injection as a `Map` of type `Map<String,Object>` but the `IEclipseContext` is not a flat structure but an hierarchical one. There's an application-context (attached to the Application-Element), a window-context (attached to a Window-Element), a Part-Context attached to a Part, … .

b) The implementation of the `SendMail` handler:

```
public class SendMail {
  @Execute
  public void sendMail(@Named(IServiceConstants.ACTIVE_SHELL) Shell shell,
    MTrimmedWindow window, IMail mail, IAccount account, IPresentationEngine engine) {

    if( mail.getTo() == null || mail.getTo().trim().length() == 0 ||
      mail.getTo().indexOf('@') == -1 ) {

      MessageDialog.openError(shell, "No Recipient", "Your mail has no recipient.");
      return;
    }

    if( mail.getSubject() == null || mail.getSubject().trim().length() == 0 ) {
      if( ! MessageDialog.openQuestion(shell, "No subject",
        "You have not set a subject would you like to proceed?") ) {
        return;
      }
    }

    account.getSession().sendMail(account, mail);

    // Bug in 4.0 we need to tear down through the presentation-engine
    engine.removeGui(window);
    window.getParent().getChildren().remove(window);
  }
```

```
  @CanExecute
  public boolean canExecute(@Optional IAccount account, @Optional IMail mail) {
    return account != null && mail != null;
  }
}
```

The only interesting thing is the use of the IPresentationEngine which we need to use because of another problem with removing already existing workbench-windows from the application in the 4.0 codebase.

Since we have to use the IPresentationEngine, we should briefly explain what it is and does. The IPresentationEngine is the one who translates the Application-Model into an UI and reflects all modifications you make on the model (like the one we made in NewMailHandler) in the UI - ranging from easy ones like updating the Label to such complex ones like adding new windows, moving views and much more.

We can now run our application which looks like this:

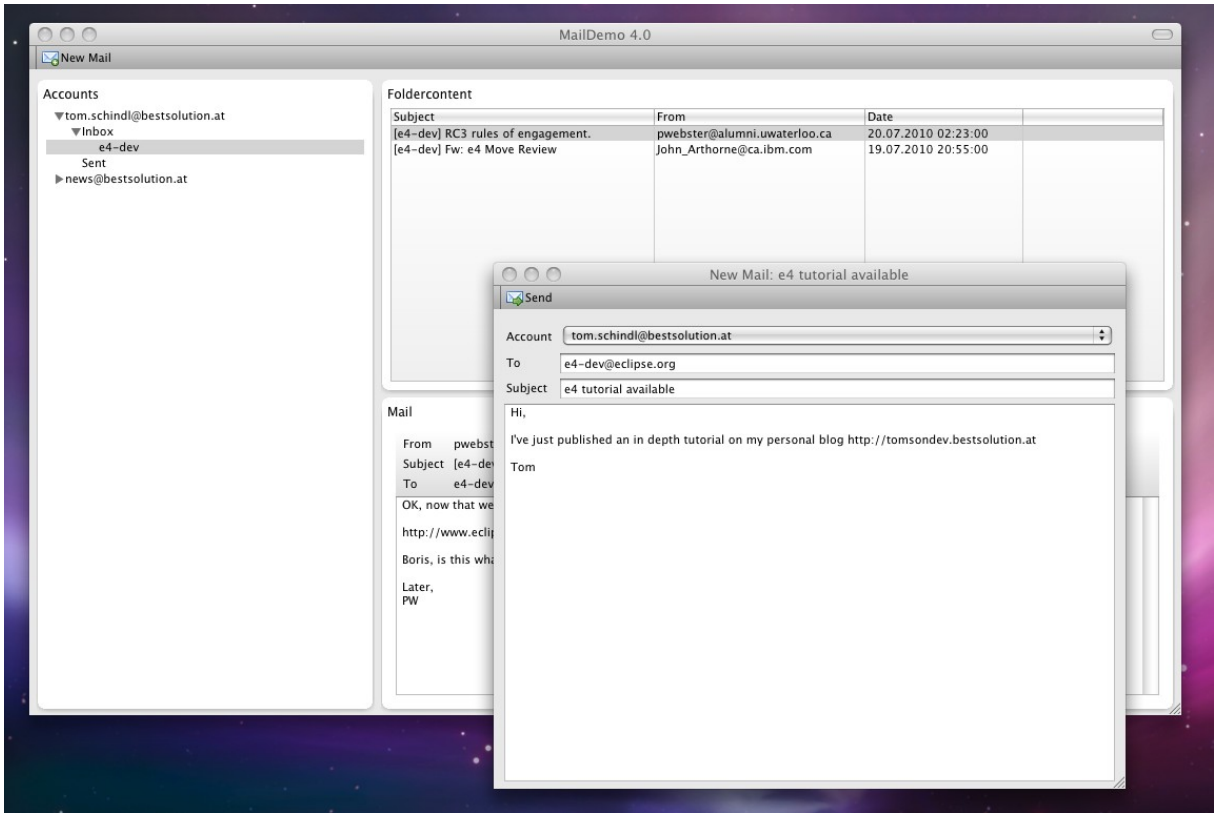


Figure 30: Final Application

You'll immediately notice a problem after having started and stopped the application once. The new mail windows we created are restored on restart so we need to take care on shutdown to remove them before the workbench state is persisted.

The only available solution in 4.0 is to install a lifecycle-handler in the using the product definition in "org.eclipse.e4.demo.mailapp".

Create a class named LifecycleHandler which looks like this:

```
public class LifecycleHandler {
  @PreSave
  void cleanupModel(MApplication application, EModelService modelService) {
    List<Object> list = modelService.findElements(application, null, null,
      Collections.singletonList("temporaryObject"));

    for (Object o : list) {
      EcoreUtil.delete((EObject) o, true);
    }
  }
}
```

There are 4 annotations available to plug into the lifecycle:

- **@PostContextCreate**: Called after the application context is created
- **@ProcessAdditions**: Called before the model is passed to the renderer
- **@ProcessRemovals**: Called before the model is passed to the renderer
- **@PreSave**: Called before the model is persisted

The other nice thing is that the e4 provides us a services to interface with the application model and we are using it here to remove objects tagged with "temporaryObject" still we need some EMF-Help to clean the elements (note this is the first time and only time in this tutorial that you see that the application model is defined using EMF)

Finally we need to register the lifecycle handler in our product-definition which looks 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>
         <property
```

```
               name="lifeCycleURI"
               value="platform:/plugin/org.eclipse.e4.demo.mailapp/org.eclipse.e4.demo.mailapp.Life
cycleHandler">
         </property>
      </product>
   </extension>

</plugin>
```

Improve Usability with Keybindings

A professional application allows the user to use the application using the keyboard and so we add as a final polish step keybindings.

We can only define keybindings for actions triggered through commands (so we can't add one for our Connection Configuration dialog which is launched using a `DirectMenuItem`).

The first thing we need to create are binding contexts which define under which circumstance a keybinding is active.

a) Open the `Application.e4xmi` and check the "Root Context" checkbox

b) We set the "Id" to "`org.eclipse.e4.demo.mailapp.app.context`" and the Name "`In the Application`"

c) Create a child context with "Id" set to "`org.eclipse.e4.demo.mailapp.mainwindow.context`" and the Name to "`In Main Window`".

d) Bug workaround: There's a small bug in the Binding-Implementation which yields an exception if no "org.eclipse.ui.contexts.dialog" is available. So we add a dummy context with this id.

Now we have to define a two `BindingTables` which we connect to the contexts defined before.

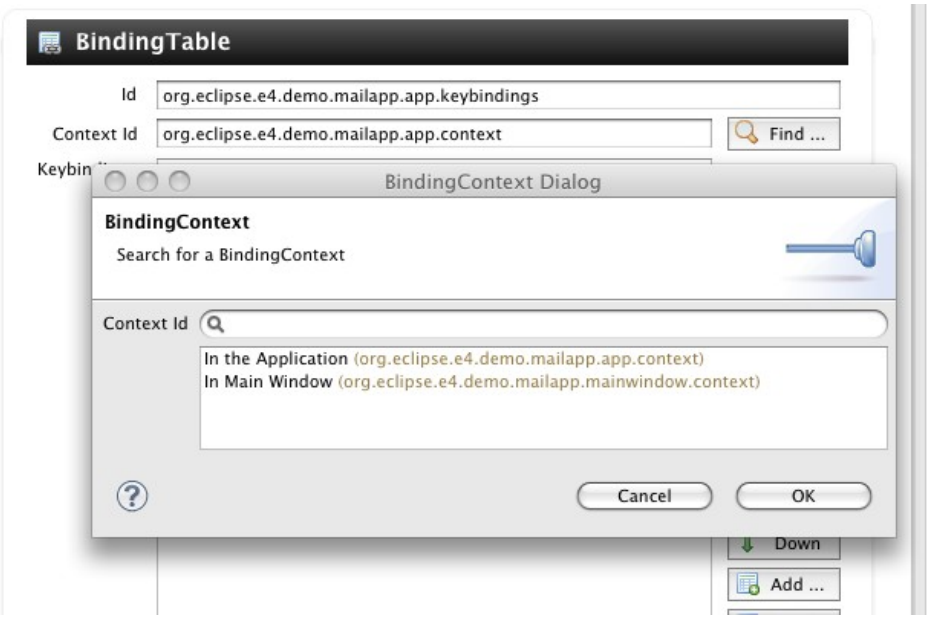


Figure 31: Connect BindingTable to Contexts

- BindingTable 1:
  - Id: `org.eclipse.e4.demo.mailapp.app.keybindings`
  - Context Id: `org.eclipse.e4.demo.mailapp.app.context`
- BindingTable 2:
  - Id: `org.eclipse.e4.demo.mailapp.mainwindow.keybindings`
  - Context Id: `org.eclipse.e4.demo.mailapp.mainwindow.context`

Now we are ready to create keybindings. The first one we create is a special one to shutdown the application (CTRL+Q/CMD+Q).

You should create the following Model-Elements:

- A Command:
  - Id: `org.eclipse.e4.demo.mailapp.command.exit`
  - Name: Exit Application
- A Handler:
  - Command: `org.eclipse.e4.demo.mailapp.command.exit`
  - Class URI pointing to: `org.eclipse.e4.demo.mailapp.ExitHandler`
- A Separator to the File-Menu
- A HandledMenuItem to the File-Menu with
  - Id: `org.eclipse.ui.file.exit` (this is important for OS-X users because using this id ensures that the menu entry is moved to the

Application-Menu)

- Label: Exit
- Command: `org.eclipse.e4.demo.mailapp.command.exit`

The application model looks like this after you have executed the actions above:

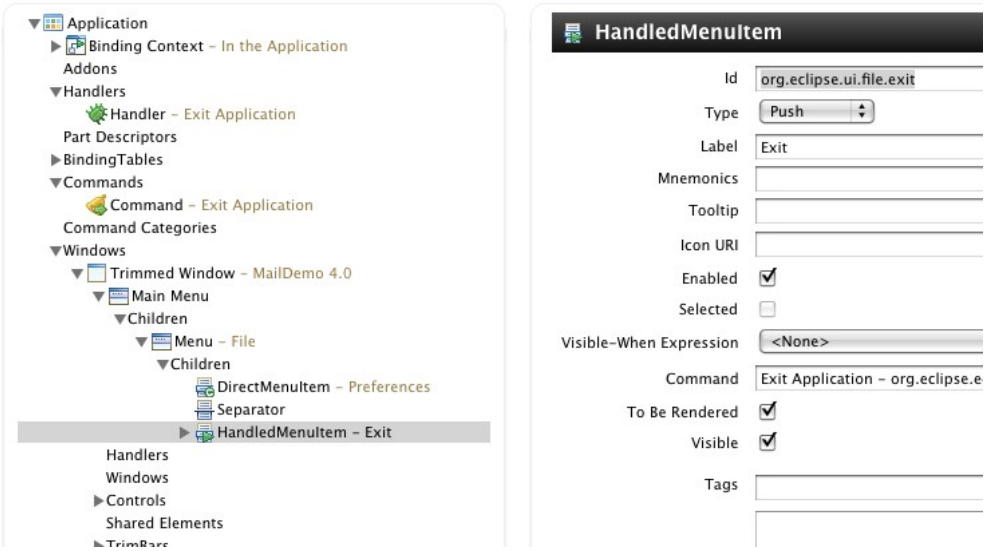


Figure 32: Model Structure

The next model change is to add the KeyBinding to the "org.eclipse.e4.demo.mailapp.app.keybindings"-BindingTable. You'll have to set the following values on the element:

- Sequence: M1+Q
- Command: `org.eclipse.e4.demo.mailapp.command.exit`

Last step in our model is to

- associate our Application-Element with the "`org.eclipse.e4.demo.mailapp.app.context`"-Context.
- Associate our TrimmedWindow with the "`org.eclipse.e4.demo.mailapp.mainwindow.context`"-Context

Open the elements and enter the above ids into the "Binding Context"-Field and press "Add".

Finally we implement the ExitHandler class:

```
public class ExitHandler {
  @Execute
  public void execute(@Named(IServiceConstants.ACTIVE_SHELL) Shell shell,
    IPresentationEngine engine) {

    if( MessageDialog.openQuestion(shell, "Exit Application?",
```

```
            "Do you really want to exit the application?") ) {

        engine.stop();

      }
    }
}
```

You can now launch your application and hit CTRL/CMD+Q and should be prompted with dialog like this:

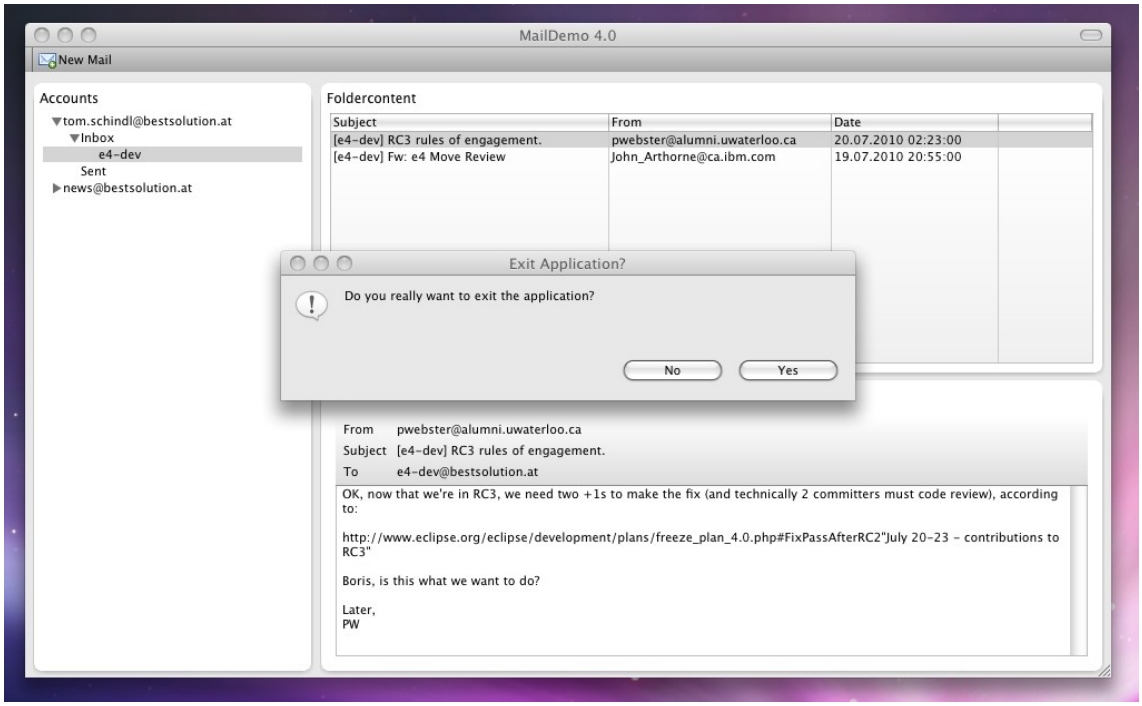


Figure 33: Exit Application Dialog

Last step in the binding story is to contribute a Keybinding for the "New Mail"-Command.

Open the `fragement.e4xmi` and add a new "`String Model Fragment`" with the following attributes:

- Element `Id`: `org.eclipse.e4.demo.mailapp.mainwindow.keybindings`
- Featurename: `bindings`

Add a KeyBinding-Element to the elements list and set the following values:

- Sequence: `M1+N`
- Command: `org.eclipse.e4.demo.mailapp.command.newmail`

Want the source code? Look at "Source Zip for Contribution" on page

57.

Get the source

Using git-repository at github.com

The sources used in this document are all available from git repository hosted at http://github.com/tomsontom/e4demo/ .

You'll can install EGit-Plug-in into your 4.0 SDK to check clone the respository to your local workspace and import the sources.

Download zip-Files

- Premade Service-Bundle: http://tomsondev.com/e4rcp/services-bundle.zip
- Source Zip for Chapter 2: http://tomsondev.com/e4rcp/chapter2.zip
- Source Zip for Chapter 4: http://tomsondev.com/e4rcp/chapter4.zip
- Source Zip for Chapter 5: http://tomsondev.com/e4rcp/chapter5.zip
- Source Zip for Chapter 6: http://tomsondev.com/e4rcp/chapter6.zip
- Source Zip for Chapter 7: http://tomsondev.com/e4rcp/chapter7.zip
- Source Zip for Chapter 8: http://tomsondev.com/e4rcp/chapter8.zip
- Source Zip for EventSystem: http://tomsondev.com/e4rcp/eventsystem.zip
- Source Zip for Contribution: http://tomsondev.com/e4rcp/contribution.zip

Closing words

I hope this RCP-Application tutorial has helped introduce you to the most important concepts behind this new platform and provide you a jump start on writing e4 applications.

There is more though as we have actually only scratched the surface of this new platform. Here are some topics we haven't yet looked into:

- Customization of the Workbench-Rendering by writing our own Renderers
- Writing our own `IPresentationEngine` to e.g. use a different Widget-Toolkit like Swing to render the Workbench-UI
- Extending the Workbench Model to introduce your own concepts

- CSS-Themes
- Advanced model contribution using Java-Code

I'd like to remind you once more that the 4.0 release of the SDK is marked as an “Early Adopter Release” and not meant for daily use as your IDE.

The e4-runtime is a new application framework you can use to write UI-Applications. Similar to the 4.0 SDK you must know that you are running on a bleeding edge platform which has not yet released a stable API nor is bug free.

Still we think if you're starting of with a new RCP-Application you should consider using e4 as an option but it depends on a case by case base if this makes sense for you.

All API provided currently by e4 is provisional and can change in the 4.1 timeframe so if you decide to use e4 as your application framework you should subscribe to the e4-dev-mailing list.

The provisional state of the API should not solely regarded as a disadvantage as you have the opportunity to influence its direction as an early adopter.

We need your *constructive* feedback to make e4 the most powerful UI-Application-Framework