Eclipse 4 Goes Formal: API You Can Rely On

- Eric Moffatt
  - Platform UI and e4 committer
  - Eclipse 4 Dev Lead
  - “Pixel Pusher” I focus on the UI bits; renderers, Min/Max...
What do we use API for anyway?

- Getting new stuff into the UI...
  - Views / Editors
  - Menus / Toolbars and their Items
  - Trim Controls
  - Perspectives

- ... and getting it to work
  - Accessing services
  - Listening for changes in UI state
Eclipse 3 API (aka 'Old School')

- Organically grown over many years
- Single points of access means inheriting lots of dependencies you really didn't need
  - IWorkbench[Window | Page] are used as access points meaning that while you only want to consume one service you end up dragging in every type referenced.
- Uses singletons (i.e. PlatformUI)
  - Bad for multi-instance apps...
Eclipse 4 API

- Two general mechanisms cover everything
  - UI Model (How you talk to us)
    - Defines what you see on the screen
    - Sends events for all model changes
  - Dependency Injection (How we talk to you)
    - You just describe what you need
    - How you access services
    - Dynamic re-injection (replaces most listeners)
    - You describe what you require using annotations...but only what you need
package apidemo.handlers;

import org.eclipse.core.commands.AbstractHandler;

/**
 * The constructor.
 */

public class APIDemoHandler extends AbstractHandler {

    /**
     * the command has been executed, so extract extra information from the application context.
     */

    public APIDemoHandler() {
    }
}
package apidemo.handlers;

import org.eclipse.core.commands.AbstractHandler;

/**
 * Our sample handler extends AbstractHandler, an IHandler
 * @see org.eclipse.core.commands.IHandler
 * @see org.eclipse.core.commands.AbstractHandler
 */

public class APIDemoHandler extends AbstractHandler {

/**
 * the command has been executed, so extract extra
 * from the application context.
 */

MTrimBar(s)
package apidemo.handlers;

import org.eclipse.core.commands.AbstractHandler;

/**
 * Our sample handler extends AbstractHandler, an IHandler.
 * @see org.eclipse.core.commands.IHandler
 * @see org.eclipse.core.commands.AbstractHandler
 */

public class APIDemoHandler extends AbstractHandler {

  /**
   * The constructor.
   */

  public APIDemoHandler() {
  }

  /**
   * the command has been executed, so extract extra information from the application context.
   */
}
package apidemo.handlers;

import org.eclipse.core.commands.AbstractHandler;

/**
 * Our sample handler extends AbstractHandler, an IH
 * @see org.eclipse.core.commands.IHandler
 * @see org.eclipse.core.commands.AbstractHandler
 */

public class APIDemoHandler extends AbstractHandler {
    /**
     * The constructor.
     */
    public APIDemoHandler() {
    }

    /**
     * 
     * @param event
     */
    public void execute(Ex...
Simple Scenario Demo

- Standard (3.x) Command to add a new view to the 'bottom' stack
  - Uses new API to manipulate the model
  - New MyPart is an e4 part
    - It's a POJO
    - Gets its info through DI
    - Listens to both model and preference changes
What's Next?

- Docs...Docs...Docs !!
- Loosen up extension point 'class' restrictions
- Finish up LifeCycle events
- Start blurring the Eclipse 4 / E4 boundary
  - Start breaking some Views and services away from depending on the Workbench
A Start on the Docs

Questions ?
A Start on the Docs

public class APIDemoHandler extends AbstractHandler {
    /**
     * The constructor.
     */
    public APIDemoHandler() {
    }

    /**
     * the command has been executed, so extract the needed information
     * from the application context.
     */
    public Object execute(ExecutionEvent event) throws ExecutionException {
        IWorkbenchWindow window = HandlerUtil.getActiveWorkbenchWindowChecked(event);
        MApplication theApp = (MApplication) window.getService(MApplication.class);
        if (theApp != null) {
            IWindow activeWin = theApp.getSelectedElement();
            EModelService ms = activeWin.getContext().get(EModelService.class);
            MPartStack btmStack = (MPartStack) ms.find("bottom", activeWin);
            MPart curPart = (MPart) ms.find("APIDemo.myPart", btmStack);
            if (curPart == null) {
                curPart = ms.createModelElement(MPart.class);
                curPart.setId("APIDemo.myPart");
                curPart.setCloseable(true);
                curPart.setLabel("API Demo");
                curPart.setContributionURI("bundleclass://APIDemo/apidemo.MyPart");
                btmStack.getChildren().add(curPart);
            }

            EPartService ps = activeWin.getContext().get(EPartService.class);
            ps.activate(curPart);
        }

        return null;
    }
}
@Focus void setFocus() {
    label.setFocus();
}

@Inject @Optional void tbrHandler(@UIEventTopic(UIEvents.UIElement.TOPIC_WIDGET) Event eventData) {
    Object changedElement = eventData.getProperty(UIEvents.EventTags.ELEMENT);
    if (!(changedElement instanceof MPart))
        return;

    visCount = countVisibleElements();
    updateMessage();
}

@Inject void animationsPrefChanged(@Preference(nodePath="org.eclipse.ui", value="SHOW_MEMORY_MONITOR") String value) {
    showHeapValue = value == null ? "false" : value;
    updateMessage();
}

@Inject void activePartChanged(@Named(IServiceConstants.ACTIVE_PART) MPart newPart) {
    apLabel = newPart == null ? null : newPart.getLabel();
    updateMessage();
}

@Inject void selChanged(@Optional @Named(IServiceConstants.ACTIVE_SELECTION) Object newSel) {
    selCount = 0;
    if (newSel instanceof IStructuredSelection) {
        selCount = ((IStructuredSelection)newSel).size();
    }
    updateMessage();
}