Yennick Trevels

- Home (/)
- Blog (/blog)
- Contact (/contact.html)

BLOG

JavaFx: Structuring Your Application - Overview

Oct 15th, 2013

JavaFx is a new rich client framework by Oracle. It allows you to create visually pleasing enterprise business applications with the Java language. While many small demos are showing up in the community, there aren't many articles on how to structure your JavaFx application. Using a good structure makes it easier to maintain your application, add new features and improves overall stability.
These are all very important in enterprise applications. This serie of articles will try to remedy that by showing you one of many ways to structure your application. This is certainly not the only way to do it, but it will give you a starting point. Even this structure is open for improvement (as every structure is).

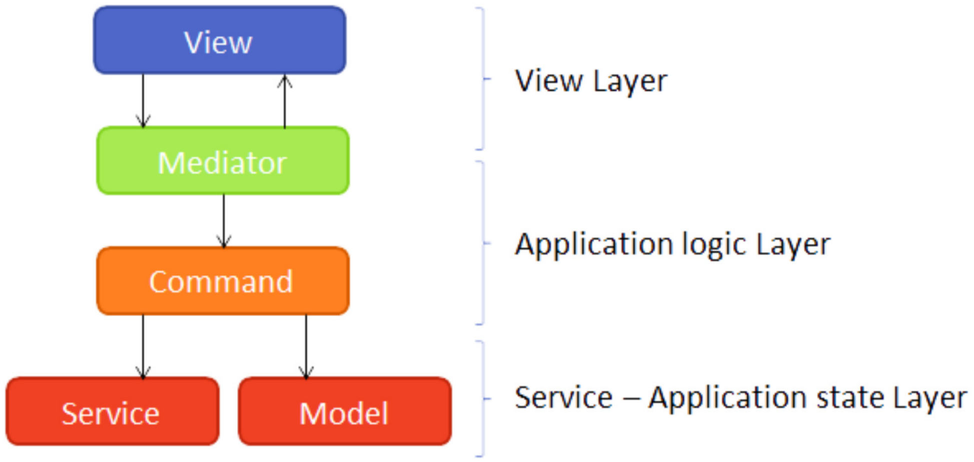
This serie will consist of four blog posts:

1. Overview (this blog post)
2. View layer
3. Application logic layer
4. Service - Application State layer

High-level Overview

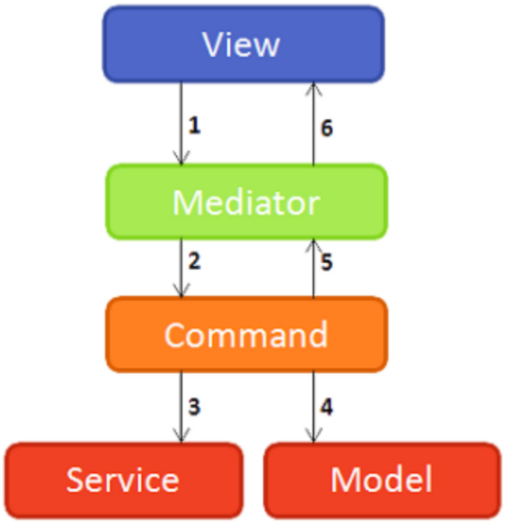
The structure I will describe is based on Robotlegs, an Apache Flex micro-architecture library. But instead of using events to decouple the layers, I went with for direct calls (with interfaces in-between). While events allow you to decouple the layers even more, I prefer direct calls with interfaces over events because they make it easier to follow the flow of your application.

The diagram below shows the different components and layers in my architecture. The arrows indicate the dependencies.



View: This is where the JavaFx components are used to build the UI.
Mediator: The mediator facilitates the communication between the view and the application logic layer. This is why it's both included in the view layer as the application logic layer. This layer should not contain application logic, it's just a postman who receives calls from the view and delegates them to the application logic layer and vise versa.
Command: This is where your application logic will reside. A Command is a class which executes a single unit of work and will be disposed when it's finished.
Service: A service is used to access external resources. This can be a remote service, a file on the filesystem or a device attached to the user's pc.
Model: A model stores the state of your application. This can be the selected record which should be accessible in multiple views.

The diagram below shows a sample flow of an application. In this case the user clicked on a button to delete a contact in a list.



1. The view asks the mediator to delete the item by calling the deleteContact(contact) method on the IContactListMediator interface.
2. The mediator creates a new DeleteContactCommand and registers a success handler on it. Then it calls the start() method on the command.
3. The command calls the contact service to delete the contact from the database
4. The command calls a method on the model to remove the contact from the application state. It receives the updated list of contacts from the model
5. The command finishes and triggers the registered success listeners. It provides the updated list of contacts to the success handlers.
6. The success handler in the mediator asks the view to update the list of contacts based on the updated contact list it got from the command.

Used libraries

For my application I didn't use any framework (except for JavaFx, obviously), I wanted to create my own structure without the overhead of a framework.
All I used was the JavaFx framework and Guice for dependency injection.
For those interested in build tools, I used the JavaFx Gradle plugin created by Danno Ferrin to create a build script.

Dependency Injection

MAPPING YOUR DEPENDENCIES

A DI library like Guice allows you to inject instances of certain classes into other classes without you having to manually instantiate and inject them. This can greatly improve the quality of your code since you don't have to write all that yourself. Guice is basically the glue that holds your application together.

In Guice you have to create a Module class which defines all the mappings. Such a module class can look like this for a simple application:

```java
public class EventManagerModule extends AbstractModule {

    @Override
    protected void configure() {
        bind(EventManagerModule.class).toInstance(this);

        mapViews();
        mapMediators();
        mapCommands();
        mapServices();
        mapModels();
        mapInfrastructure();
    }

    private void mapViews() {
        bind(IEventOverviewView.class).to(EventOverviewView.class);
        bind(IEventListView.class).to(EventListView.class);
        bind(IEventDetailView.class).to(EventDetailView.class);
    }

    private void mapMediators() {
        bind(IEventOverviewMediator.class).to(EventOverviewMediator.class);
        bind(IEventListMediator.class).to(EventListMediator.class);
        bind(IEventDetailMediator.class).to(EventDetailMediator.class);
    }

    private void mapCommands() {
        bind(ICommandProvider.class).to(CommandProvider);

        bind(LoadEventsCommand.class);
        bind(AcceptEventCommand.class);
        bind(DeclineEventCommand.class);
    }

    private void mapServices() {
        bind(IEventService.class).to(StubEventService.class);
    }

    private void mapModels() {
        bind(IEventSelectionModel.class).to(EventSelectionModel.class).in(Singleton.class);
    }

    private void mapInfrastructure() {
        bind(ITranslationProvider.class).to(ResourceBundleTranslationProvider.class).in(Singleton.class);
    }
}
```

So in most cases you will be mapping implementations to interfaces. Always try to use interfaces. This allows you to, for example, create a stub implementation of a service (in this case StubEventService) and a real implementation which you can then just swap by changing the Guice mapping. Only for the Commands I didn't create interfaces because they wouldn't add that much value. They would all practically be marker services, but you'll see that in my following posts.

While this Module implementation is ok for a small application, it can quickly explode in one huge class when you're building a bigger application. This is where bootstrap classes come in handy. A bootstrap class is just a class which initializes a certain part of your application. In this case it would initialize the mappings of the views, the mediators or the commands. Such a bootstrap class would look like this:

```
public class MediatorMapper implements Mapper {

    private AbstractModule module;

    public MediatorMapper(AbstractModule module) {
        this.module = module;
    }

    public void bootstrap() {
        module.bind(IEventOverviewMediator.class).to(EventOverviewMediator.class);
        module.bind(IEventListMediator.class).to(EventListMediator.class);
        module.bind(IEventDetailMediator.class).to(EventDetailMediator.class);
    }
}
```

Your module would then look like this:

```
public class EventManagerModule extends AbstractModule {

    @Override
    protected void configure() {
        bind(EventManagerModule.class).toInstance(this)

        new ViewMapper(this).bootstrap();
        new MediatorMapper(this).bootstrap();
        new CommandMapper(this).bootstrap();
        new ServiceMapper(this).bootstrap();
        new ModelMapper(this).bootstrap();
        new ModelMapper(this).bootstrap();
    }
}
```

WIRING IT ALL TOGETHER

Now how do you wire all this together? Well, that is what I'm going to show you now.

In your Main class you will create a Guice Injector and with this injector you will instantiate the main view of your application. Your Main class will look like this:

```java
public class Main extends Application {

    public static void main(String[] args) {
        Application.launch();
    }

    @Override
    public void start(Stage stage) throws Exception {
        Injector injector = Guice.createInjector(new EventManagerModule());
        EventOverviewView eventOverviewView = injector.getInstance(EventOverviewView.class);

        Scene scene = SceneBuilder.create()
                .root(
                    eventOverviewView.getView()
                ).width(900)
                .height(700)
                .build();
        scene.getStylesheets().add(Main.class.getResource("/EventManager.css").toExternalForm());

        stage.setScene(scene);
        stage.show();
    }
}
```

The views that are being used in the main view are also injected in the main view. Also, the mediator that is behind the main view is also injected. This patterns is used for every view.

```java
public class EventOverviewView implements IEventOverviewView {

    @Inject
    public IEventOverviewMediator mediator;
    @Inject
    public EventListView eventListView;
    @Inject
    public EventDetailView eventDetailView;

    ...
}
```

Then, in the mediator, we inject the view and the commandProvider (this is a class which allows to create command instances on the fly).

```java
public class EventOverviewMediator implements IEventOverviewMediator {

    @Inject
    public IEventOverviewView view;
    @Inject
    public CommandProvider commandProvider;

    ...
}
```

And in the application logic layer we inject the service/model in the command.

```java
public class LoadEventsCommand extends Service<List<EventVO>> {

    @Inject
    public IEventService eventService;

    ...
}
```

Conclusion

In this post I gave a broad overview of the structure and explained how dependency injection fits into this.
In the following posts I will go into more detail for each layer and give some best practices.
You can find the full sample application here: https://github.com/SlevinBE/JavaFx-structure-demo (https://github.com/SlevinBE/JavaFx-structure-demo)

Posted by Yennick Trevels Oct 15th, 2013 JavaFx (/blog/categories/javafx/)

Tweet (http://twitter.com/share)
« Automated Gradle project deployment to Sonatype OSS Repository (/blog/2013/10/11/automated-gradle-project-deployment-to-sonatype-oss-repository/) JavaFx: Structuring your Application - View Layer » (/blog/2013/10/15/javafx-structuring-your-application-view-layer/)

Recent Posts

- My 2013: Achievements Unlocked! (/blog/2013/12/29/my-2013-achievements-unlocked-slash/)
- My View On: Functional Programming Principles in Scala Course (/blog/2013/11/29/my-view-on-functional-programming-principles-in-scala-course/)
- JavaFx: Structuring your Application - Service and Application State layer (/blog/2013/10/15/javafx-structuring-your-application-service-and-application-state-layer/)
- JavaFx: Structuring your Application - Application Logic Layer (/blog/2013/10/15/javafx-structuring-your-application-application-logic-layer/)
- JavaFx: Structuring your Application - View Layer (/blog/2013/10/15/javafx-structuring-your-application-view-layer/)

Receive my advice & experiences right in your inbox

first name email address

Subscribe

- (https://twitter.com/YennickT)
- (http://www.linkedin.com/in/yennicktrevels)
- (https://github.com/SlevinBE)