VAADIN TRAINING

Organizing your code

Overview

Software patterns are tools. As with any tool, using it correctly will yield better results. To pick the correct tool, you need to familiar with multiple alternatives. In most projects, one pattern is not enough. Be prepared to combine and modify patterns where needed.

There is no silver bullet pattern for Vaadin that will work in all cases. That said, many projects share commonalities, such as Main Layout, Navigator and Event Buses.

Main Layout

The Main Layout concept means that you have a centralised place where you handle navigation.

The main layout often builds the UI root, and is a Layout itself. The Main layout creates and positions the menu, header, footer, and main content area.

The Main Layout is usually created after the User has logged in, to save on resource during startup:

JAVA

```
public class MyUI extends UI {
  protected void init(VaadinRequest request){
  etContent(new LoginView());
}

public void loginSuccessful(User user){
  setContent(new MainLayout(user));
  }
}
```

The Main Layout then creates the structure of the application, and reacts when navigation is needed:

JAVA

```
public MainLayout(User user){
  addComponent(new Header(user));
  addComponent(new Menu(user));
  mainContentWrapper = new Panel();
  addComponent(mainContentWrapper);
  navigateTo(new DashBoardView());
}

public void navigateTo(View view){
  mainContentWrapper.setContent(view);
}
```

Navigator

The Navigator is a built-in tool in Vaadin to help you with changing one part of your application out for another when the user wants to navigate. It uses URI fragments, which enable bookmarking views and parameters, and also allows using the browsers back and forward-features.

When you use Vaadin, it is recommended you group your layouts and components into logical groups representing Views. Examples of a View is e.g. `InboxView`, `UserListingView` and `ComposeEmailPopupView`. Each of these are logical wholes, and the user moves from one to another; they have no common components.

The View can consist of the whole visible area of the UI, but more typically excludes the fixed parts such as a menu and a header. As Vaadin can easily replace only a part of the UI, it makes sense not to include the duplicate menus and such with every View, but keep them separate. You 'navigate' from one view to the other by removing and adding the view to a main layout, like this:

JAVA

```java
public void navigateTo(View newView){
  mainLayout.removeComponent(currentView);
  mainLayout.addComponent(currentView = newView)

  // optional
  newView.setSizeFull();
  mainLayout.setExpandRatio(newView, 1);
}
```

The Navigator actually has a View interface for this purpose. Each view you want to use with the Navigator should implement it:

JAVA

```java
public class DashboardView
  extends CustomComponent
  implements com.vaadin.navigator.View {
    ...
}
```

The interface has only one method, `enter()`. The method is called whenever the Navigator makes that view visible, and is a good place to e.g. load new data from the backend:

JAVA

```java
public class MyView extends VerticalLayout implements View {

  @Override
  public void enter(ViewChangeEvent event) {
    // called every time this view is made visible
    refreshTableData();
  }

  ...
}
```

To use the Navigator in your application, you need to do two things in addition to creating the View classes. You need to create a Navigator object for your UI, and you need to register the views you want to use with view names. When creating the Navigator, you decide where in your UI the Views will be placed. The Navigator constructor takes two parameters, where the first one is always the UI, and the second is where you want the Views to be placed:

JAVA

```java
public Navigator(UI ui, ComponentContainer container)
public Navigator(UI ui, SingleComponentContainer container)
public Navigator(UI ui, ViewDisplay viewDisplay)

// Example if the second method above:
public class NavigatorUI extends UI {

  @Override
  protected void init(VaadinRequest request) {
    ...
    Navigator navigator = new Navigator(this, this);
    ...
  }
}
```

The example code above would always render each View directly under the UI itself, meaning that the whole UI area is consumed by the View. Most of the time you want to render the View as a sub-section of your UI, because you have menus, headers, etc. visible as well. Thus the second parameter can be any Panel or Layout you want (see Main Layout example later in this document). A `ViewDisplay` alternative is also available, giving you full control over where and how to place views.

When you've created the Navigator, you need to register views. You do this with the `addView()` method in Navigator, that takes a String identifier for the View and a reference to the view itself:

JAVA

```java
public class NavigatorUI extends UI {

@Override
protected void init(VaadinRequest request) {
  Navigator navigator = new Navigator(this, this);

  navigator.addView(“dashboard”, new DashboardView());
  // Or this:
  navigator.addView(“dashboard”, DashboardView.class);
  }
}
```

Both methods above bind the `DashboardView` class to the identifier "dashboard", but there is a difference in behaviour. The first method, that receives a View object, keeps that object in memory and re-uses it while navigating. This means any changes you make inside the view will be remembered when the user navigates back to this view.

In the second example, we do not pass an object but the Class of the view. In this case, the Navigator will create a new `DashboardView` object on each navigation, and discarding the old one. This means the View is always rebuilt from scratch. Whichever you should use depends on the view and whether the view should remember settings or not. In either case, the `enter()`-method is called every time the view is made visible.

When you have created the Navigator and registered your Views, you can very easily navigate from one View to the other by calling this method:

JAVA

```java
navigator.navigateTo("dashboard");
// or, if you don't have the Navigator reference handy,
UI.getCurrent().getNavigator().navigateTo("dashboard");
```

This tells Navigator to replace the current view with the dashboard and to change the URI fragment to "dashboard" so that the browser knows we've changed 'pages'. The URI looks like this:

`http://localhost:8080/MyApp/#!dashboard`

where:

`#!` : Navigator identifier
`dashboard` : View name

In addition to allowing the user to user the browsers back- and forwards features, the URI fragment can be stored in a bookmark. When creating the Navigator in `UI.init()`, the Navigator will read the fragment and automatically navigate to the correct view.

Navigator also supports adding a parameter String to Views when navigating. You can use this to e.g. automatically load selected data in the `enter()`-method:

JAVA

```
// When we call this
navigator.navigateTo("dashboard/" + user.getId()+"/edit");

// The resulting URI is this:
http://localhost:8080/MyApp/#!dashboard/42/edit

#! : Navigator identifier
dashboard : View name
42/edit : Parameter String

// And we can get the parameter in enter():
public class MyView extends VerticalLayout implements View {
  @Override
  public void enter(ViewChangeEvent event) {
    String params = event.getParameters(); // "42/edit"
    // load user with id 42 from the DB and enable editor
  }
}
```

Our Main Layouts usually uses the Navigator:

JAVA

```
public MainLayout(){
  ...
  Navigator nav = new Navigator(UI.getCurrent(), mainContentWrapper);
  registerViews();
  nav.navigateTo(DashboardView.VIEW_NAME); // String constant to avoid typos
}

public void navigateTo(String viewId){
  nav.navigateTo(viewId);
}
```

How we organize project code

At Vaadin, we like the beauty of simplicity. We rarely over-design anything up front, but let the structure grow over time. This doesn't mean that we start without structure or let it grow uncontrollably; we simply start from a simple, well defined place and update the architecture when necessary. In the beginning we usually decide on the following:

- Main Layout, since navigation needs designing immediately
    - Decide if we use the Navigator or go custom-made (e.g. multi-level)
- Services tech; do we use EJBs, Spring or something else, including Data
- Packaging structure for View files
    - Views typically communicate through an EventBus
- Any utils we need; Security, Translations, Logging, etc.

As an example, we could base the stack on CDI and JavaEE, Hibernate (JPA) and DeltaSpike data, allowing us to use the CDI event bus in our views. The packaging structure typically looks something like this:

```
com.acme.superspeedapp.data
com.acme.superspeedapp.services
com.acme.superspeedapp.ui
com.acme.superspeedapp.ui.main
com.acme.superspeedapp.ui.views.login
com.acme.superspeedapp.ui.views.dashboard
com.acme.superspeedapp.utils
```

Custom components go either in the View package they are used (if it's only one view) or in a shared package:

`com.acme.superspeedapp.ui.views.login.RememberMeCheckBox` // only used in login view

or

`com.acme.superspeedapp.ui.components.TimeSelector` // can be used anywhere

What about patterns? We used to be proponents of MVP, but we've backed off a bit. The reason is not that MVP would be a bad choice, but that full-blown MVP is too complicated for many cases. This only creates boilerplate code without actually helping the developers, so we've started using simpler methods for organizing our UI code. Nowadays, we concentrate on _naming_, _code conventions_, and _migration rules_ instead of a single pattern.

We've found that these three things control the evolution of any pattern. If you have good conventions (and really follow them), you can start with very simple patterns and evolve into quite complex setups. More importantly, the rules gives your code readability because they follow the same progression everywhere. Based on naming, any coder knows what classes do what, regardless if the View consists of one or five classes.

How to evolve the structure of a View class (and package)

It's fine to start with all View code in a single class, at first. Business login should be separated into their own methods, that do not do UI changes. It is OK to access the backend from the business methods. Pay attention to method names; it should be clear immediately what a method does. For example, init() is a bad name, loadAndSortData() is better. Data is sent as simple method parameters from one to the other.

When you class grows beyond a certain point (we use number of lines, around 500 or so) it's time to refactor the class. You do this by splitting your single View class into two; one handles visuals and the other handles data. The name of the new class should relate to the View, and have the same suffix as all another similar classes (e.g. Presenter, Controller, or Logic). Simply move all business methods from the View to the new class and you're done. The View and Logic classes can refer directly to each other, like so:

JAVA

```java
public class DashboardView {

  DashboardLogic logic;
  public DashboardView() {
    logic = new DashboardLogic(this);
  }
}

public class DashboardLogic {

  DashboardView view;
  public DashboardLogic(DashboardView view) {
    this.view = view;
}

// Example method that can be called from the View
public void refreshTableData() {
  data = backend.getData();
  dataProvider = packageAsDataProvider(data);
  view.setTableDataSource(dataProvider);
  }
}
```

The general flow of calls between the View and Logic classes goes like this:

1. The user clicks a Button and the `ClickListener` in the view is invoked by the Framework
2. The `ClickListener` calls the presenter to do something, such as save or update data
3. The presenter calls backend methods to accomplish the operation and possible load new data
4. The presenter finally calls a method in the View to update data

Further updates to the classes are discussed after Event Buses.

Event Buses

Event buses let different parts of the UI communicate indirectly with others. For example, a View might want to update some data in the application header, such as a user name. We could give the View a direct reference to the Header object and have it know what method in the header to call. But this will make the View dependent on the Header class, which is bad practice. Using an Event Bus, we can break the connection.

Event Buses work by having a centralized location where actors can fire and listen to events. In our example, the View is an Actor that wants to fire an Event, and the Header is an actor who wants to listen to them. Instead of the classes referring to each other, both only refer to the Event Bus. The View now has no idea what happens after the event is fired (as it shouldn't), and the Header doesn't know from where the event was fired (which it shouldn't). As a bonus, we can very easily fire the same `UsernameUpdatedEvent` from any other class, and also add classes that react to the same event, such as a menu class or logging utility.

If you use event buses, be aware that firing events inside an UI is OK (for instance from View to Header or MainLayout), but events between different UIs (or even user sessions) are dangerous. There are many pitfalls when doing this, and you might mess up your application state, transaction scopes and others. Similarly, the more complex Event chains you use, the harder they are to debug. We typically use Event Buses for one-off messages between views or from views to the top level (e.g. navigation and logout events), but not inside single view packages. This gives us nice separation between different modules of the application, but is still easy enough to follow when debugging. If you find your Views to constantly send messages between each other, maybe the View and Navigator concept isn't the right one for you.

Isolating the View from data model changes

In the examples above we haven't mentioned the data classes at all. This is by design; in the simple pattern we use, we don't want extra classes complicating things so we use backend data entities directly. In most cases this is fine, but when your data model grows more complex you might want to check out options of isolating your Views from the actual data layer.

The idea is that you create bespoke data model classes for views. This is especially beneficial if your view only uses a small prtion of a large data entity, or it uses data from multiple entities. If you have a bespoke class, it makes it easier to use the data in your view.

There are two options for creating bespoke data classes; creating a Proxy and creating a Data Transfer Object, DTO. Both have getters and setters to important data, to present a good API to the View. The difference is how the real data from the backend is used. With Proxies, you store a reference to the backend data and proxy all method calls to the right underlying data:

JAVA

```
public class PersonProxy {
  tprivate PersonEntity person;

    public PersonProxy(Person person) {
      this.person = person;
      }

    public void setName(String name) {
      this.person.setName(name);
      }

    public String getName() {
      return this.person.getName();
    }
    ...
}
```

With a DTO, you don't store a reference, but instead copy over the relevant data when you create the DTO:

JAVA

```
public class PersonDTO {
  private String name;

  public void setName(String name) {
    this.name = name;
    }

  public String getName() {
    return name;
  }
  ...
}
```

Using a DTO makes sense for instance if you are using JPA, and you want to avoid lazy loading exceptions; with a DTO you load everything you need in one back end call, and there are no JPA proxy references left in your data to cause issues.

Using a dedicated Model class

Many UI patterns revolve around a Model object (such as Model-View-Presenter and Model-View-Controller). In our pattern above we specifically ignore a separate Model object, as we don't need one. For more complex cases it might make sense to create such a class however. Again, there are multiple options on how to create a Model class. One option is to move data manipulation from the Logic class to the Model, letting it access the backend. That doesn't really work with our pattern however, as it would break the evolution progress but suddenly having a completely unrelated class accessing the backend.

We can still have use of a dedicated Model class though. For instance, we can create a class that is a data holder for all the data we need in the View, and that is capable of detecting changes to the data and notifying any listeners about it. We call this the Intelligent Model pattern. Here is an example for how the Model automatically notifies the View that data has been updated:

JAVA

```java
public class DashboardView implements DashboardModelListener {

  public DashboardView() {
  DashboardModel model = new DashboardModel();
  DashboardLogic logic = new DashboardLogic(this, model);
  model.registerListener(this);
  }
}

public class DashboardLogic {

  public void viewEntered() {
    List<Persons> list = Backend.getPersons();
    model.setPersons(list);
    // no need to call View, it already knows
    }
}

public class DashboardModel {

  private List<Persons> persons;
  private DashboardModelListener listener;

  public void setPersons(List<Persons> list) {
    persons = list;
    if(listener != null) {
      // let listener know that data changed
      listener.updatePersons(persons);
      }
    }
}
```

The Intelligent Model could then be expanded to handle e.g. data conversions, validation, and backend transactions, depending on the complexity of the View.

A edicated Model is also a good idea if you want to share data between multiple Views, such as in a wizard type UI.