

Background

During the course, we will improve a given application, step by step. We'll start with the stub code, and build on that during the day. If you get stuck, there is a solution for each exercise separately in the solutions folder. For exercises 2 and 3 you can copy the solutions 'navigator' and 'refactor' to have a clean slate for both exercises if you wish.

Exercise 1: Navigation

The first exercise is to implement navigation between views. Create a Main Layout class so that it uses the Navigator API to handle the views and navigation between them.

Things you'll need to do

1. Create a MainLayout class in the root package. The class should extend HorizontalLayout.
   a. The main layout will be the UI content (it *is* rootLayout); move all code from the UI that handles layouting to MainLayout, to the constructor.
   b. MainLayout should also implement ClickListener, move that code from the UI.
   c. When you have moved everything, there should be one line left in ExampleApp.init(): `setContent(new MainLayout());`
   d. Test your code at this stage; it should look and work identically to the starting code.
2. There are two views in the application, make the views Navigator compatible by implementing the *com.vaadin.navigator.View* -interface
   a. Add static view name string to the classes
3. In the MainLayout class, create a navigator instance and register views to the navigator. The Panel should be the destination of the views.
   a. AuditingView should be registered with .class, DepartmentView with an object.
   b. Store the Navigator in a class field
4. Change the click listeners in the MainLayout class so that clicking on a button will transfer the user to the corresponding view using the Navigator
5. The navigateToDefaultView method should use the Navigator too
6. Update any possible data in the views using the newly created enter() method.
   a. In DepartmentView, move the data fetch to the enter() method from the constructor

Bonus task: The URL to access the department view is http://localhost:8080/exercise-mvp/#!department
It is possible give parameters to views which can then be handled in the view's enter method. Your task is to check, if a person ID is given as a view parameter, then the person with that ID should automatically be selected from the table for editing. It is also good design to scroll the grid to the selected value. There is a method in the Service to get a single Person based on ID.

Vaadin Docs:
Navigator: https://vaadin.com/docs8/-/part8/framework/advanced/advanced-navigator.html

Exercise 2: Presenters (Logic classes)

The customer has requested you to refactor their application to have better maintainability. You'll start this task by refactoring application logic into presenters.

Improve code compartmentalizing and testability by separating UI code and -logic from each other. Views should still listen to events from the UI components but the presenter will decide what should be done when an event happens. Presenters are also the entry point to other subsystems like services.

Things you'll need to do:
1. Create a presenter class for the Audit view, copy and paste the code below for the presenter and view.
2. Create a presenter class for the Department view.
   a. The view should create its own presenter upon construction and give a reference to itself in the presenters constructor, see AuditingView code.
3. The contents of the view should be refreshed upon navigation
4. Move the business logic from the view implementations to the presenters.
   a. See below for suggested API for the DepartmentPresenter
   b. DepartmentPresenter should handle at least:
      - Fetching employees from service and updating the view with them upon navigation
      - Handle the saving of a *Person*-entity through the service

When using the Vaadin Navigator API, one can easily separate the navigation event from the view creation event. If the view is given as a class-reference to the navigator, the view will be recreated upon each navigation. However, if an instance of the view is given to the navigator the state of the view is preserved. We should try to make our views so that either mode can be used. This is why we should only create the components of the view in the constructor and fill/refresh the data of the view on the *enter*-event.

Below is illustrated how the enter pattern can be used in the Auditing view.

```
public class AuditingPresenter {
      private AuditingView view;

      public void setView(AuditingView auditingView) {
            view = auditingView;
      }

      private void fetchInitialData() {
            for (String message : AuditLogService.getAuditLogMessages()) {
                  view.addAuditLog(message);
            }
      }

}

public class AuditingView extends VerticalLayout implements View{


      public final static String VIEW_NAME = “auditing";
```

```
	private AuditingPresenter presenter;
	private CssLayout messageLayout;

	public AuditingView() {
		final Label header = new Label("Audit Log Messages");
		header.setStyleName(ValoTheme.LABEL_H2);
		addComponent(header);
		setMargin(true);

		presenter = new AuditingPresenter();
		presenter.setView(this);
	}

	@Override
	public void enter(ViewChangeEvent event) {
		presenter.fetchInitialData();
	}

	public void addAuditLog(final String message) {
		messageLayout.addComponent(new Label(message));
	}

}
```

The Audit view should be registered with the class alternative, so it is re-created every time the user navigates to it.

Suggested DepartmentPresenter API:

```
public class DepartmentPresenter {

	private final DepartmentView view;

	private final PersonService service = new PersonService();
	private List<Person> employees;

	public DepartmentPresenter(DepartmentView view) { ... }

	public void requestSave(Person person) { ... }

	public void viewEntered(String parameters) { ... }
}
```

Exercise 3: Separate models from entities

In this exercise we will change the DepartmentView so that it uses more lightweight objects for the listing view by creating a DTO for the Person class.

Things you'll need to do:

1. Create an PersonListingDTO class in the data package which will function as the lightweight model for a Person entity
   a. The class should have the following fields; id, first name, last name, email.
2. In PersonService, implement the `getListingDTOs()` method
   a. for each Person, create a DTO instead, and return a list of the DTOs
3. Change the DepartmentView Grid to use your new DTO class instead of Person
4. The DepartmentPresenter should fetch the DTOs instead of Persons
   a. Change the `getEmployees()` call to `getListingDTOs()`
5. What about the EmployeeEditor? It should continue using the full Person class, not the DTO. We do need a few additions though.
   a. Refactor the Grid selection listener so that on selection, the view notifies the Presenter this DTO was selected.
   b. The Presenter should fetch the Person object from the Service based on the person id
   c. The Presenter then tells the View to open the editor with that object.
   d. The view opens editor and sets the fragment as before
6. You can enable more fields in the form by setting `showHiddenFields` to `true` in EmployeeEditor for a demo on how the entity can have much more data than what is visible in the Grid.

Tip: When creating the DTO make sure you name the getters and setters the same as in the Person class; this way you don't need to change he Grid column code at all.