

**Router API:** 

# Enabling Routing and Navigation



#### Vaadin training set

#### **Vaadin Foundation**

- Introduction
- Layouting
- Creating Forms
- Data Lists with Grid
- Routing and Navigation
- Theming and Styling Applications



#### Agenda

- Part 1:Router basics
- Part 2: Navigation API
- Part 3: Application Layout
- Part 4: Navigation Lifecycle

Each part has an exercise.



## Router API, Part 1

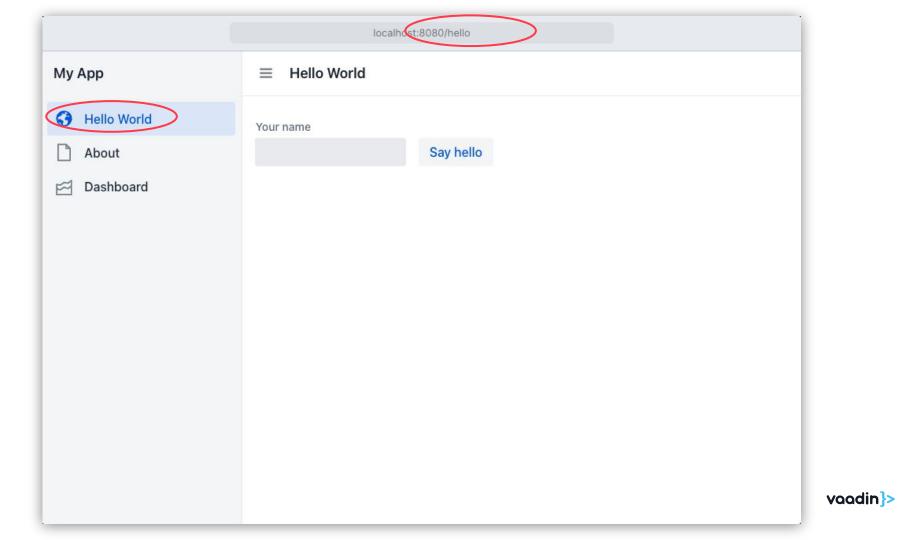
**Router basics** 

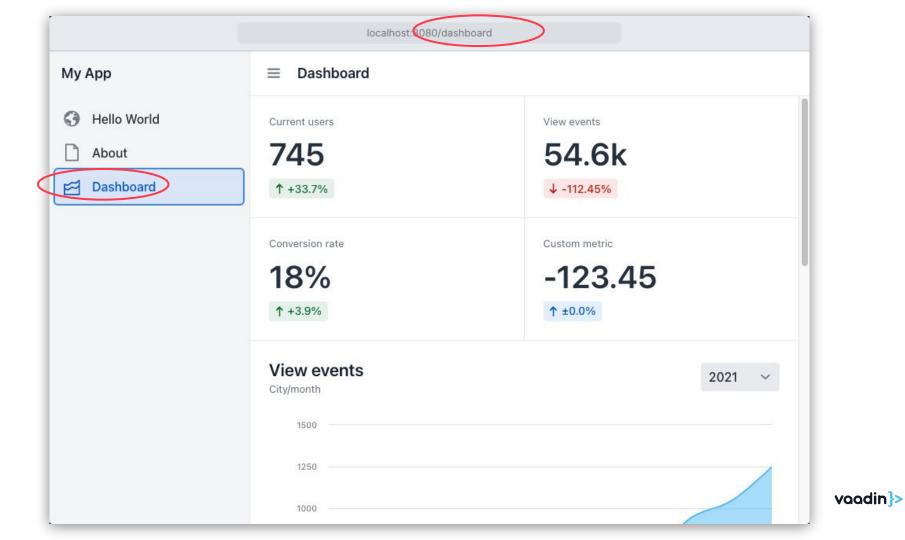


### https://yourdomain.com/app/view

**Application URL** 

Route





## The Router API enables navigation and linking for your web app:

- Navigating to a specific view and data using URLs ("deeplinking")
- 2. Updating browser's address bar

3. Keep registry of available views

## How to enable routing?

Routing can be enabled by adding the @Route annotation to any component class definition:

```
@Route("home")
public class HomeView extends VerticalLayout {}
```

@Route takes a String parameter for the desired URL path, e.g.

- @Route("home")
- @Route("some/path")
- @Route("") this is the root (and default) path

What if you want multiple URLs to point to the same class?

localhost:8080

localhost:8080/home

What if you want multiple URLs to point to the same class?

localhost:8080

localhost:8080/home

Additional routes can be added with **@RouteAlias** 

```
@Route("")
@RouteAlias("home")
public class HomeView extends VerticalLayout {
     ...
}
```

**URL parameters?** 

### https://yourdomain.com/app/view/12345/...

**Application URL** 

Route

Parameter(s)

### **Enable URL Parameters handling**

To accept URL parameters, the component needs to implement the **HasUrlParameter** interface.

#### **URL Parameters for navigation targets**

This example demonstrates the concept:

```
@Route("greet")
public class GreetingComponent extends Div implements HasUrlParameter<String> {
    @Override
    public void setParameter(BeforeEvent event, String parameter) {
        setText(String.format("Welcome %s!", parameter));
    }
}
```

If you navigate to the address "appdomain.com/greet/vaadin" the result is the message "Welcome vaadin!". The parameter is mandatory - trying to navigate to "appdomain.com/greet/" will not work without extra configuration.

#### **Optional Parameters**

The parameter can be defined as optional by using @OptionalParameter in the method. Not providing a parameter in the URL and not having @OptionalParameter will not match the route class which expects a parameter.

#### **Wildcard Parameters**

In case more parameters are wanted (e.g. greet/one/two/three or greet/42/edit), the URL parameter can also be marked as a wildcard with @WildcardParameter. Only String type wildcards are supported:

What about query parameters?

#### Query Parameters for navigation targets

The router will accept these parameters by default, but will not do anything with them:

http://example.com/products?name=laptop&color=red

#### Route templates

To access even more complex URL patterns, you can use Route Templates. URL parameters don't have to be at the end of the string - you can use templates like /users/:userld/edit, where the :userld is the parameter variable. More complex template patterns and regular expressions are also supported - the following would also be a valid route template:

product/:identifier/:category?/resource/:id([0-9]\*)/:path\*

#### Using Route Templates

Route templates are defined in the Route value string, so they won't be used through the HasUrlParameter interface. Instead, you access them in a BeforeEnter navigation event listener. We'll cover navigation lifecycle events in more detail later.

```
@Route("user/:userID/edit")
public class UserProfileEdit extends Div implements BeforeEnterObserver {
    private String userID;

    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        userID = event.getRouteParameters().get("userID").get();
    }
}
```

#### Overlapping route definitions?

**More specific** route definitions override the generic ones if multiple Route definitions cover the same URL:

```
@Route("greet")
public class OptionalComponent extends Div implements HasUrlParameter<String> {
     @Override
     public void setParameter(BeforeEvent event, @OptionalParameter String parameter) { ... }
}

VS

@Route("greet/training")
public class TrainingComponent extends Div { ... }
```

When navigating to appdomain.com/greet/training, you will end up with TrainingComponent being loaded, not the OptionalComponent, even though the base URL (greet) is the same. The same logic applies for Route Templates - an exact match wins, then come required parameters and last wildcards.

## **Dynamic Routes**

Registering new routes dynamically



#### **Dynamic Routes**

In addition to using the static Route annotations, you can define any number of additional routes dynamically. This is very useful when you need to change mappings because e.g. data changes, or permissions for the user changes mid-session.

There are two classes of dynamic routes, both have their uses:

- Session scope routes Solo para el usuario actual
- Application scope routes Para todos los usuarios, independientemente del estado de la sesión

Both scopes are managed through the **RouteConfiguration** utility class. Con esta clase se registran nuevas rutas programáticamente.

#### Session scope dynamic routes

Session scope refers to the current user of the application; similar to e.g. the Spring session scope. This scope starts when a user opens the application for the first time, and ends when the user either logs out or the session closes by itself after a timeout.

Session scope routes hence only work for a single user, and will not work when the user logs in again unless they are re-registered. They are great for e.g. permission-based navigation, where you allow specific users access to specific views:

```
// Only this user gets the admin access
RouteConfiguration.forSessionScope().setRoute("admin", AdminView.class);
```

#### Session scope dynamic routes

Routes can also be removed, but you'll need to be very specific when you do this.

```
// The AdminView will not be available anymore
RouteConfiguration.forSessionScope().removeRoute(AdminView.class);
```

You can for example remove a base path while keeping more complex ones:

```
// Remove the "/users" path but keep e.g. "/users/123"
configuration.removeRoute("users", UsersView.class);
```

#### Application scope dynamic routes

The Application scope is the application uptime, from deployment to shutdown. It is independent of user sessions, meaning this scope manages routes for all users.

All @Route configurations are placed in the application scope registry. You can add and remove routes from it any time, but the best place is at application startup (before users arrive) in a ServiceInitListener:

```
public void serviceInit(ServiceInitEvent event) {
    // Example: add this view only when in development
    if (!event.getSource().getDeploymentConfiguration().isProductionMode()) {
        RouteConfiguration.forApplicationScope().setRoute("crud", DBCrudView.class);
    }
}
```

Cuidado al eliminar rutas, porque se eliminan para todos los usuarios

#### Listening to Route changes

If you change routes dynamically in your application, you might also want to update e.g. menu items when the changes occur; e.g. any RouterLinks you created before the change might now be invalid.

The RouteConfiguration class has methods to get all registered routes, as well as adding listeners for when routes change. You can use this functionality to update your app dynamically:

```
RouteConfiguration.forSessionScope().addRoutesChangeListener(event -> {
    List<RouteBaseData<?>> addedRoutes = event.getAddedRoutes();
    List<RouteBaseData<?>> removedRoutes = event.getRemovedRoutes();
});
```

#### Route lookup order

Since there are a few different ways to create routes, in what order are they evaluated?

Session Routes override Application routes:

http://yourdomain.com/users/42



#### **Session scope routes**

- That exact route?
- 2. Partial route with mandatory parameter?
  - 3. Matching wildcard route?



#### **Application scope routes**

- 1. That exact route?
- 2. Partial route with mandatory parameter?
  - 3. Matching wildcard route?



Excepción Not found

# Router error and exception handling

# Exception-based routing error views

You can define specific Components to handle routing-time Exceptions of certain types with the HasErrorParameter interface.

These work as navigation targets (just as components with @Route), but instead of having a specific URL, they are <u>loaded</u> as needed.

```
public class ProductErrorView extends
    VerticalLayout implements
    HasErrorParameter<ProductNotFoundException> {
```

throw new ProductNotFoundException();

# Exception-based error views

Exception type (class) is checked first, after which super types are checked.

```
public class ProductErrorView extends
      VerticalLayout implements
      HasErrorParameter<<pre>ProductNotFoundException> {
 public class GenericErrorView extends
       VerticalLayout implements
       HasErrorParameter<RuntimeException> {
              . . .
```

# Exception-based error views

Specific Exception types may have only one handler class. This is checked when the app starts.

```
public class ProductErrorView extends
     VerticalLayout implements
     HasErrorParameter<ProductNotFoundException> {
          ...
```

```
public class AnotherView extends
    VerticalLayout implements
    HasErrorParameter<ProductNotEoundException> {
```

#### Router-specific error views

The Router API includes a few bespoke Exception classes for handling common cases with the same method as in the generic case, such as a basic 404 when trying to navigate to an unknown path:

# Changing the Page Title

#### Static page titles

For simple, non-changing page titles, you can use the PageTitle annotation:

```
@PageTitle("Home Page")
@Route("home")
class HomeView extends Div {
         HomeView(){
             setText("This is the home view");
         }
}
```

#### Dynamic page titles

If the title depends on data and you need to update it HasDynamicTitle is your friend. The getPageTitle() method is called after the component is attached.

```
@Route(value = "blog")
class BlogPost extends Component implements HasDynamicTitle {
    private BlogPostData currentPost;

    @Override
    public String getPageTitle() {
        return "Blog: " + currentPost.getTitle();
    }
}
```

### Dynamic page titles

In addition, you can update the page title anytime using the following snippet:

```
UI.getCurrent().getPage().setTitle("42");
```

### Summary, Part 1

- How to enable routing
- URL parameters
- Dynamic Routes
- Router error handling
- Changing the page title



# **Exercise 1**

**Enable Routing in your application** 

### Router API, Part 2

Navigation: Adding navigation into your application

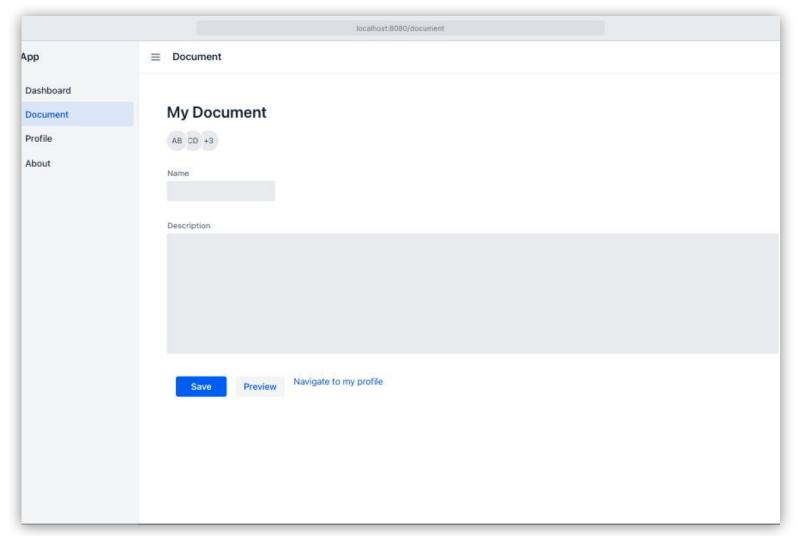
Como el usuario navega a través de la aplicación



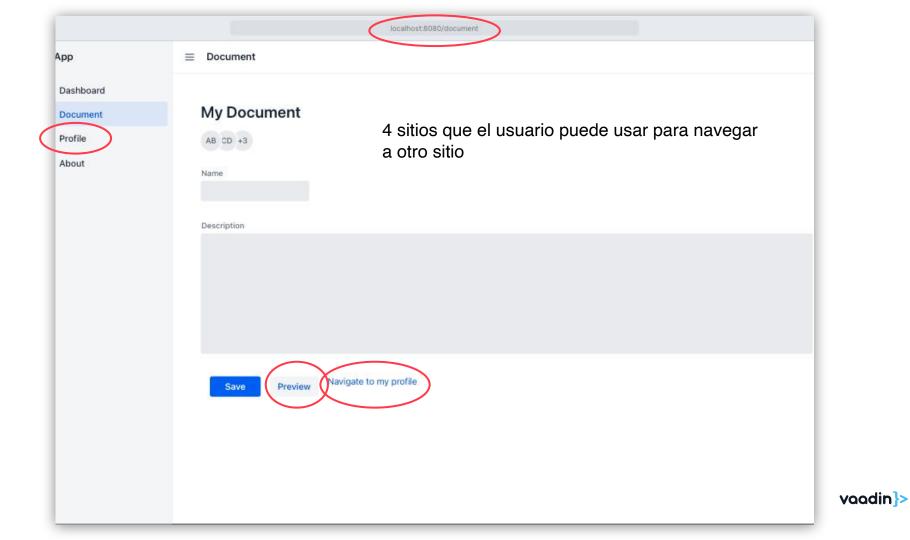
### Recap, part 1

- How to enable routing
- URL parameters
- Dynamic Routes
- Router error handling
- Changing the page title









The easiest way to create navigation links in your UI is to use the RouterLink component:

This will give you a normal HTML link that takes you to the given class. The Router will parse out the correct path from the Route annotation for you.

Using a RouterLink instead of an Anchor (normal link) gives you the <u>advantage of not</u> having a <u>full page reload</u>, as RouterLink understands that this is an in-app link. Conversely, RouterLinks do not work on external addresses, you should use Anchor for those.

If you need more control over navigation (e.g. giving parameters) or don't want to use RouterLinks to trigger navigation, then you can use the UI.navigate() API.

Es una navegación programática

If you need more control over navigation (e.g. giving parameters) or don't want to use RouterLinks to trigger navigation, then you can use the Ul.navigate() API.

If you need more control over navigation (e.g. giving parameters) or don't want to use RouterLinks to trigger navigation, then you can use the Ul.navigate() API.

As RouterLinks are plain <a> elements in the browser, they are also friendly to assistive technology like screen readers. For this reason, you should prefer RouterLinks over "invisible" navigation whenever possible.

(o botones de navegación)

#### Generating URLs programmatically

When you need to generate or modify a route, you can get any of the preconfigured route Strings from the Router class:

```
@Route("path")
public class PathComponent extends Div { ... }

// returns 'path'
String route = RouteConfiguration.forSessionScope().getUrl(PathComponent.class);
```

Funciona porque la session Scope hereda la application Scope

### Generating URLs programmatically

The API supports parameters, of course:

```
@Route("path")
public class PathComponent extends Div implements HasUrlParameter { ... }

// returns 'path/test'
String route = RouteConfiguration.forSessionScope().getUrl(PathComponent.class, "test");
```

### Generating URLs programmatically

Note that the Router only returns the path inside the app deployment context; to get the full path, you can use VaadinServletRequest:

StringBuffer url = VaadinServletRequest.getCurrent().getRequestURL();

#### Passing data between views in Java

Most of the UI.navigate methods used in routing <u>return the actual instance of the target view</u> as an Optional. This allows the possibility of communicating between your views in Java:

```
@Route("edit")
public class UserEditor extends Div {
   public void editUser(User user) {
        // do something with user
   }
}
```

#### Passing data between views in Java

Note that using the <u>server-side data</u> passing mechanism doesn't automatically update any URL parameters; if you want your views to be deep-linkable, you'll need to do that yourself.

```
public void editUser(User user) {
    // do actual UI changes
    createFormForUser(user);
    // update the URL
    updateUrlParameters(user);
}

private void updateUrlParameters(User user) {
    // generate the wanted URL, in this case with the user ID parameter
    String deepLinkingUrl = RouteConfiguration.forSessionScope().getUrl(getClass(), user.getId());
    // updating does not reload the page or cause navigation
    getUI().get().getPage().getHistory().replaceState(null, deepLinkingUrl);
}
```

#### **Adding Query parameters**

You can use the QueryParameters through a helper class which takes in a Map:

```
Button button = new Button("Navigate to my view");
button.addClickListener(e -> {
          Map<String, List<String>> parameters = new HashMap();
          parameters.put("param", Arrays.asList("value"));
          QueryParameters queryParameters = new QueryParameters(parameters);
          button.getUI().ifPresent(ui -> ui.navigate("myView", queryParameters));
});
```

#### Adding Query parameters

And you don't need to use the hardcoded string:

```
Button button = new Button("Navigate to my view");
button.addClickListener(e -> {
          Map<String, List<String>> parameters = new HashMap();
          parameters.put("param", Arrays.asList("value"));
          QueryParameters queryParameters = new QueryParameters(parameters);
          String viewUrl = UI.getCurrent().getRouter().getUrl(MyView.class);
          button.getUI().ifPresent(ui -> ui.navigate(viewUrl, queryParameters));
});
```

### How to receive the query parameters?

You can use **BeforeEnterEvent** or **AfterNavigationEvent** to parse the incoming query parameters. We'll discuss about this in Part 4.

### Summary, Part 2

- Navigation
- Generating URLs
- Passing data between views in Java
- Programmatic Query parameters



# **Exercise 2**

**Add navigation** 

### Summary, Part 2

- Navigation
- Generating URLs
- Passing data between views in Java
- Programmatic Query parameters



## Router API, Part 3

**Application Layout** 



### Recap, parts 1-2

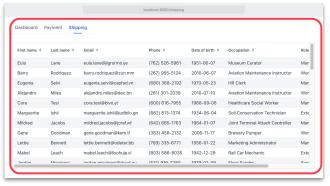
- How to enable routing
- URL parameters
- Navigation
- Passing parameters and data between views

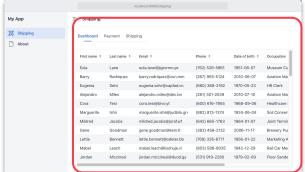


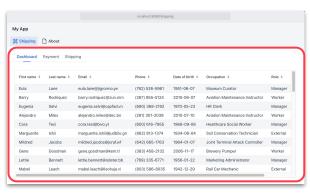
When defining routes using **@Route("path")**, the component will, by default, be rendered inside the **<body>** tag of the page.

But what if you want a header, or a menu, visible on each page?

Hasta ahora hemos visto que se cambia siempre toda la UI, pero eso no suele ser lo que se quiere.

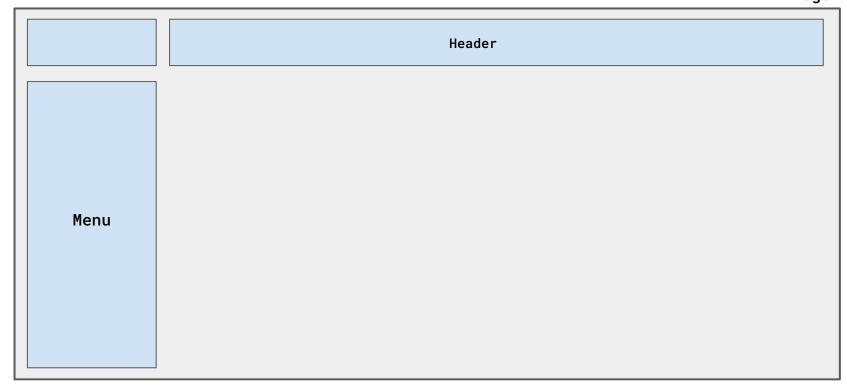






Page

Page



Page Header View content area Menu

#### En terminología Vaadin

### Managing the Application Layout

MainLayout



#### **Parent Layouts**

A parent Layout can be defined with the layout attribute in the @Route annotation:

When navigating between components that use the same parent class, the parent will be re-used will not update during navigation.

#### **Parent Layouts**

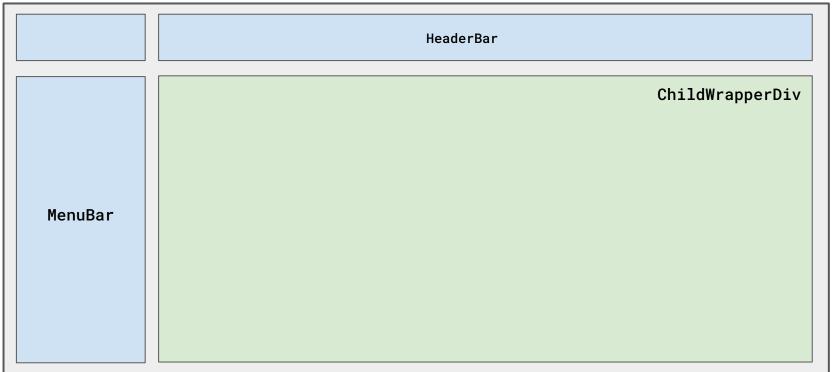
Any component used as a parent layout must implement the RouterLayout interface:

```
public class MainLayout extends Div implements RouterLayout {
    ...
}
```

#### Parent Layouts implement RouterLayout

RouterLayout includes a default (but overridable) method showRouterLayoutContent(), which will appends the child view at the end of its own content:

# Parent Layouts implement RouterLayout MainLayout



# Parent Layouts implement RouterLayout MainLayout



## Parent Layouts implement RouterLayout

You can override the method for full control over where you place the child element:

# Multi-level layouting

# Multi-level layouting (nesting menus, etc)

App	■ Main Viev	W					
Main View	Dashboard	Payment	Shipping				
About							
	First name \$	Last name \$	Email \$	Phone \$	Date of birth \$	Occupation \$	Role
	Eula	Lane	eula.lane@jigrormo.ye	(762) 526-5961	1951-06-07	Museum Curator	Mana
	Barry	Rodriquez	barry.rodriquez@zun.mm	(267) 955-5124	2010-06-07	Aviation Maintenance Instructor	Work
	Eugenia	Selvi	eugenia.selvi@capfad.vn	(680) 368-2192	1970-05-23	HR Clerk	Mana
	Alejandro	Miles	alejandro.miles@dec.bn	(281) 301-2039	2010-07-10	Aviation Maintenance Instructor	Work
	Cora	Tesi	cora.tesi@bivo.yt	(600) 616-7955	1968-09-06	Healthcare Social Worker	Mana
	Marguerite	Ishii	marguerite.ishii@judbilo.gn	(882) 813-1374	1934-06-04	Soil Conservation Technician	Exter
	Mildred	Jacobs	mildred.jacobs@joraf.wf	(642) 665-1763	1964-01-07	Joint Terminal Attack Controller	Mana
	Gene	Goodman	gene.goodman@kem.tl	(383) 458-2132	2006-11-17	Brewery Pumper	Work
	Lettie	Bennett	lettie.bennett@odeter.bb	(769) 335-6771	1956-01-22	Marketing Administrator	Mana
	Mabel	Leach	mabel.leach@lisohuje.vi	(803) 586-8035	1942-12-29	Rail Car Mechanic	Exter
	Jordan	Miccinesi	jordan.miccinesi@duod.gy	(531) 919-2280	1979-02-09	Floor Sander	Supe
	Marie	Parkes	marie.parkes@nowufpus.ph	(814) 667-8937	1939-12-11	Rock Dust Sprayer	Exter
	Rose	Gray	rose.gray@kagu.hr	(713) 311-8766	1954-12-10	Medical Esthetician	Mana

## **Nested Layouts**

So far, we've built Routes and mapped them to a parent by using the **layout** parameter of the Route. What if a deeper hierarchy of nesting?

```
public class MainLayout extends Div implements RouterLayout {...}
```

```
@Route(value = "view1", layout = MainLayout.class)
public class View1 extends Div {...}
```

```
@Route(value = "view2", layout = MainLayout.class)
public class View2 extends Div {...}
```

### **Nested Layouts**

Any RouterLayout can also have a @ParentLayout to build layout hierarchies:

```
public class TopLevelLayout extends Div implements RouterLayout {...}
```

```
@ParentLayout(TopLevelLayout.class)
public class MidLayout extends Div implements RouterLayout {...}
```

```
@Route(value = "view1", layout = MidLayout.class)
public class View1 extends Div {...}
```

```
@Route(value = "view2", layout = MidLayout.class)
public class View2 extends Div {...}
```

### @ParentLayout annotation

There are no restrictions on the amount of nested layouts.

Note: Normally, a class implementing RouterLayout doesn't have its own @Route annotation since it only a holder for other views (that have their own @Route).

Any class using the **@ParentLayout** annotation should also be a **RouterLayout** or an error view (implements **HasErrorParameter**)

# Summary, Part 3

- Application Layout
- Parent Layouts
- Multi-Level Layouting



# **Exercise 3**

**Application Layout** 

# Summary, Part 3

- Application Layout
- Parent Layouts
- Multi-Level Layouting



# Router API, Part 4

The Navigation Lifecycle



## Recap, parts 1-3

- How to enable routing
- URL parameters
- Navigation
- Passing parameters and data between views
- Application Layout
- Parent Layouts
- Multi-Level Layouting



## The Navigation Lifecycle

During a navigation event, three events are fired and can be reacted on:

- BeforeEnterEvent → BeforeEnterObserver
- BeforeLeaveEvent → BeforeLeaveObserver
- AfterNavigationEvent → AfterNavigationObserver

Para afinar la lógica de la aplicación y hacer algo más especial al navegar entre rutas Order of the navigation events:



#### BeforeEnterEvent

The BeforeEnterEvent is fired before the view is rendered.

This event is typically used to reroute / forward navigation requests:

- Check login
- Handle errors

The event can be caught in any Component by implementing the BeforeEnterObserver interface.

## BeforeEnterEvent: Reroute dynamically

```
// Events can be observed not only in @Route classes, but also parent layouts
public class MainLayout extends Div implements RouterLayout, BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        if (!isUserLoggedIn) {
            event.rerouteTo(LoginView.class);
        }
    }
}
```

## BeforeEnterEvent: Reroute dynamically

You can also reroute to an error view registered for a particular Exception type:

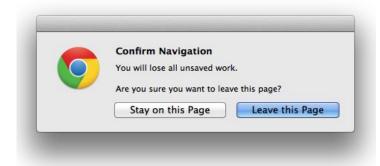
```
public class AuthenticationHandler implements BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        Class<?> target = event.getNavigationTarget();
        if (!currentUserMayEnter(target)) {
                  event.rerouteToError(AccessDeniedException.class);
        }
    }
}
```

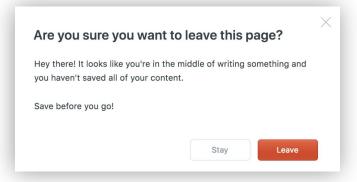
#### BeforeLeaveEvent

The **BeforeLeaveEvent** is fired when a component is removed from the DOM tree (when the Router knows that we should be navigating to another route).

- Clean up view data
- Postpone navigation ('are you sure you want to leave' functionality.)

The event can be caught in the Component by implementing the **BeforeLeaveObserver** interface.





#### BeforeLeaveEvent: Confirm leave

```
@Route("signup")
public class SignupForm extends Div implements BeforeLeaveObserver {
  @Override
  public void beforeLeave(BeforeLeaveEvent event) {
      if (this.hasUnsavedChanges()) {
         ContinueNavigationAction action = event.postpone(); // Save navigation action for later
         Dialog confirmDialog = new Dialog();
         Button confirmButton = new Button("Confirm", e -> {
            action.proceed(); // User said ok, continue with the navigation (after this listener is done)
            confirmDialog.close();
         });
         confirmDialog.add(new Text("Are you sure you want to leave?"), confirmButton);
         confirmDialog.open();
```

## AfterNavigationEvent

Fired during navigation, when the new view is in place and there can not be any more redirects. Useful for updating the UI with the final state:

- updating a separate menu item when we know which item is active.
- Handle incoming QueryParameters

The event can be caught in the Component by implementing the **AfterNavigationObserver** interface.

# AfterNavigationEvent: Dynamic page customization

```
public class SideMenu extends Div implements AfterNavigationObserver {
    Anchor blog = new Anchor("blog", "Blog");

    @Override
    public void afterNavigation(AfterNavigationEvent event) {
        boolean active = event.getLocation().getFirstSegment().equals(blog.getHref());
        blog.getElement().getClassList().set("active", active);
    }
}
```

# AfterNavigationEvent: Parsing query parameters

This listener is also a good place to parse the Query parameters and react on them:

# Summary, Part 4

- Navigation Lifecycle
- BeforeEnterEvent
- BeforeLeaveEvent
- AfterNavigationEvent



# **Exercise 4**

**Setup login** 

# Summary, Part 4

- Navigation Lifecycle
- BeforeEnterEvent
- BeforeLeaveEvent
- AfterNavigationEvent



## Summary

#### **Enabling Routing:**

- @Route
- URL parameters for navigation targets

#### Navigation:

- RouterLink
- Ul.navigate()
- Exception handling during routing



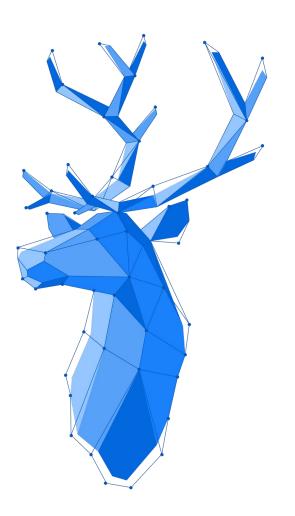
## Summary

#### **Application Layout:**

- RouterLayout
- Nested layouts with @ParentLayout

#### Navigation Lifecycle:

- BeforeEnterEvent redirect in new view
- BeforeLeaveEvent postpone in current view
- AfterNavigationEvent initialize new view



### Thank you!

