LUND UNIVERSITY

Routing and Form Validation

EDAF90 Web Programming

Per Andersson

URI and URL

- URI — identifies a resource
- URL — identifies a location
- URL is a subset of URI
- URI = scheme:[//authority]path[?parameters][#anchor]
- authority = [userinfo@]host[:port]

```
<h2 id="url">URI and URL</h2>
...
<a href="https://cs.lth.se/course/edaf90/?year=2023#url">
```

Percent encoding

URL is based on 7 bit ASCII

- safe characters: a-z, A-Z, 0-9, `-` `_` `.` `~`
- all others must be percent-encoded
- `%nn`, nn is 8 bit hexadecimal value
- percent-encoding uses utf-8
- hard to read: `https://cs.lth.se/edaf90/lecture%20notes`
- avoid non-safe characters in URL
- use `-` instead of space
- commonly not case sensitive, avoid camelCase and PascalCase
- do use kebab-kase

Routing

Routing

- the browser history is part of the user experience
- allows the user to navigate back to earlier visited pages
- an entry in the history is added when the user
  - navigates to a new page using a link
  - submits a form
- traditionally, this loads a new page from the server
- when a new page is loaded, all JavaScript objects are lost
- singel page web application prevents this using `preventDefault()` on all relevant events
- only updating the DOM will impact the user experience:
  - can not navigate using the browser history (back button)
  - can not link to inner pages

Routing Framework

- there is an API giving JavaScript direct access to the browser history
- using it manually is tedious and error prone
- let a router do the work for you
  - subscribing and manipulating the history stack
  - matching the URL to your routes
  - rendering a nested UI from the route matches

```
npm install react-router-dom
```

Link

Link

```
<Link to="/animals">animals</Link>
<Link to="animals/fish">fishs</Link>
```

- a react component
- let users navigate in your app
- clicking on it will add an entry to the browser history
- page is not fetched (`preventDefaults` on <a href="...">)
- this will update the url field in the browser
  - `<Route>` triggers re-render
- your JavaScript objects are untouched (preserve the application state)

NavLink

Add styling of active link using:

- knows if it is "active" or "pending"
- use css class to highlight active links
- `className` – normal CSS, or a function returning the css class
- by default, an `active` CSS class is added when active

```
<NavLink
  to="/messages"
  className={({ isActive, isPending }) =>
    isPending ? "pending" : isActive ? "active" : ""
  }
>
```

Link Example

```
import { Link } from 'react-router-dom';
function Menu() {
  return (
    <nav>
      <Link to="animal" />
      <Link to="animal/fish" />
      <Link to="animal/bird" />
    </nav>
  );
}
```

Route

createBrowserRouter

- renders components based on url matching
- routes are declared in a configuration object
- add to your app using the `<RouterProvider>` component

```
const router = createBrowserRouter(config);

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

Route objects

`router` is an array of Route objects
properties:

- `path` – string for match against the url
- `element/Component` – rendered when path is matched
  - `element` – an object (JSX expression)
  - `Component` – a react component (JavaScript function)
  - use one of them
- `caseSensitive` – if pattern matching is case sensitive (default fale)
- `children` – nested routes
- error handling
  - `errorElement` – an object (JSX expression)
  - `ErrorBoundary` – a react component (JavaScript function)
- and more, covered later

createBrowserRouter

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <h1>Welcome</h1>,
  }, {
    path: 'hello/world',
    caseSensitive: true,
    element: <h1>Hello World</h1>
  }, {
    path: 'about/*',
    Component: {WildcardComponent}
  },
]);
```

Path

- only one match in the array
- most constrained wins, order do not matter
- a path is composed of segments: pattern between ’/’
- text segment
  - exact match, letter by letter
  - case sensitivity is optional
  - percent decoded text, do not use `%nn`
  - use url safe characters
  - path: `/kebab-case/in-path/next-segment`
  - matches url: `kebab-case/in-path/next-segment`

Segments

- dynamic segment: starts with ':'
  - matches any characters, zero or more
  - the matched url text can be accessed in your component
  - path: `user/:id/lang/:lang`
  - matches url: `user/31/lang/se`
- optional segment, ends with '?'
  - path: `:lang?/categories`
  - matchs url: `categories`, and `en/categories`
- splats, catchall, star: ends with '/*', matches any character following the /, including other /
  - path: `files/*`
  - matchs url: `files/one/two/three`

Layout and Index routes

- Layout route
  - do not have a `path`
  - should not have siblings
  - the element/Component is always rendered, adds to the layout
  - do not consume url segments
- Index routes
  - selected if parent is matched but no siblings
  - do not have a `path`
  - matches an empty url segment
  - do have a `index` attribute

Nested Routs

Routs can be nested:

- `children` attribute of a Route object – an array of child routs
- each level matches a part of the url
- at most one `path` in the child array will be matched
- the element/Component of the matched `path` will be rendered on each level

Nested Routs

```
const routerConfig= [{
        path: "shop",
        element: <ShopFrame />,
        children: [
          {
            path: "item/:id",
            element: <Item />,
          }, {
            index: true,
            element: <ListItems />
          }]
      }, {
        path: "admin",
        element: <AdminFrame />,
        children: [
          {
```

Outlet

The child element is rendered in the parents `Outlet`

```
import { Outlet } from 'react-router-dom';

function ShopFrame() {
  return (
    <div>
      <h1>The Shop</h1>
      <ShopMenu />
      <Outlet />
    </div>
  );
}
```

Path Parameters

the router pass data from the path to the component

- specify parameters in the `path` using the syntax `:name`
- use the `useParams()` hook to get an object with the values

```
import { useParams } from "react-router-dom";

const config = {
    path="/item/:itemId",
    element={<Item />}
};

function Item() {
  let params = useParams();
  return <h2>item: {params.itemId}</h2>;
}
```

Hooks

- can be used in any child of `Route`
- `useParams()` - returns an object with the URL path parameters
- `useLocation()` - returns the browser location
- `useSearchParams()` - interact with query string in the URL
- `useNavigate()` - navigate programatically

Error handling

When exceptions are thrown in loaders, actions, or component rendering:

- the `element/Component` is not rendered
- instead the `errorElement/ErrorBoundary` is
  - `errorElement` – an object (JSX expression)
  - `ErrorBoundary` – a react component (JavaScript function)
  - use one of them
- exceptions will bubble up the router tree

Error handling

```
config = [{
  path: "/invoices/:id",
  element: <Invoice />,
  errorElement: <ErrorBoundary />
}];
function ErrorBoundary() {
  let error = useRouteError();
  console.error(error);
  return <div>Ooops! {error}</div>;
}
```

Picking a Router

For all urls belonging to the app:

- the server must return the html file bootstrapping react, `index.html`

`createBrowserRouter`

- `http://domain.se/item/42`
- node.js built in server do this for you
- configure apache with rewrites

`createHashRouter`

- `http://domain.se/#/item/42`
- compatible with all servers

React Router pre 6.4 Example

```
import { Route, Routes } from 'react-router-dom';
function App() {
  return (
    <Routes>
      <Route path="animal" element={<Animal />}>
        <Route path="fish"} element={<Fish />}/>
        <Route path="bird"} element={<Bird />}/>
        <Route index element={<SelectAnimal />}/>
      </Route>
    </Routes>
  );
}
```

/animal/fish → `<Animal><Fish /><Animal>`
/animal/cat → no match

Data API

We will return to react router data API:

- actions
- loaders
- lazy

Based on `async` functions, so we need to cover that first.

Form Validation

Form Validation

- user feedback is important
- common feedback comes from incorrectly filled forms
- takes a lot of time to implement
- html 5 introduced built in form validation

HTML 5 Form Validation

html form fields attributes, for example `<input>`:

- `required`
- `minlength` and `maxlength`
- `min` and `max`
- `type`: `number`, `email`, ...
- `pattern` a regexp

Any error prevents form submission

HTML 5 Form Validation

CSS pseudo classes set by the browser

- `:valid`
- `:invalid`

```
input:invalid {
  border: 2px dashed red;
}
input:invalid:required {
  background-image: linear-gradient(to right, pink, lightgreen);
}
.form-control:valid~.invalid-feedback {display: none;}
```

Constraint Validation API

adds read only properties to form input elements

- `validationMessage`
- `validity` a `ValidityState` object: properties `rangeOverflow`, `valid`, et.c.
- `checkValidity()`
- `setCustomValidity(message)` makes the field invalid

But, you can not style the error message

Custom Form Validation

Today, form validation is based on the following principle:

- use html 5 attributes to define requirements
- `<form novalidate>` prevents browser from displaying error messages
- validation is still carried out by the browser
- you can rely on the `:valid` and `:invalid` pseudo classes
- when needen, use JavaScript for custom form validation:
  1. catch the `submit` or `blur` event
  2. perform custom form validation, use custom CSS classes
- error messages: use css to show or hide normal html elements

```
visibility: hidden
```

```
<span class="my-error-class">invalid email</span>
```

Frameworks can help you with the details.

Bootstrap

Bootstrap have classes for styling forms and error messages:

- `<form novalidate>` to hide the browser default error messages
- html 5

```
<input maxlength="3" type="email" class="form-control">
<select required class="form-select">
<div class="valid-feedback">well done</div>
<div class="invalid-feedback">not so good</div>
```

  - different classes for different elements
  - must set the `.was-validated` class on a parent to show style/messages
- custom validation:
  - set the classes `.is-invalid` and `.is-valid` on the field element
  - warning: `:valid` is set when no html 5 requirements are specified
    all `valid-feedback` are shown
  - set `.was-validated` on the form group when mixing html 5 and custom validation in the same form

Bootstrap example

```
<form novalidate>
  <div>
    <label for="field1" class="form-label">First name</label>
    <input type="text" class="form-control" id="field1" required>
    <div class="valid-feedback"> Looks good! </div>
    <div class="invalid-feedback"> no good! </div>
  </div>
  <div>
    <button class="btn btn-primary" type="submit">Submit form</button>
  </div>
</form>
```

Security

A note on security

- client side form validation is mainly for giving users feedback
- a malicious user can always interrupt the network communication
- server can never trust data sent from the client (unless it is signed)
- always do server side data validation!
- with client side validation, server side can focus on malicious code