# Lab 7

# Web development with Elm - Intermediate concepts

## Goals

In this lab you will learn to:

1. Use text fields, checkboxes and dropdown lists
2. Understand what is REST API and how use it to obtain data
3. Use the `Http` module to make HTTP requests
4. Use the `Json` module to encode and decode JSON values
5. Write documentation for your modules
6. Write examples for your functions and test that they are correct
7. Compile Elm apps to a HTML page

## Resources

Table 7.1: Lab Resources

| Resource | Link |
|---|---|
| Elm core library | `https://package.elm-lang.org/packages/elm/core/1.0.5/` |
| Elm `Test` package | `https://package.elm-lang.org/packages/elm-explorations/test/latest/` |
| Elm `Html` package | `https://package.elm-lang.org/packages/elm/html/latest/Html` |
| Elm `Http` package | `https://package.elm-lang.org/packages/elm/http/latest/` |
| Elm `Json` package | `https://package.elm-lang.org/packages/elm/json/latest/` |

## 7.1 Input fields, checkboxes and dropdown lists

In this section we'll have a brief look at some more ways to accept input from the user. The example app is page that is used by the system administrators to create new user accounts.

### 7.1.1 Input fields

Input fields are just a field where users can type some text. Input fields can have various types, like `text` which is used for details like username, or `password` which hides the currently typed text.

They are created using the `input` tag, and we can set attributes like `type_` (`type` is reserved in Elm) and `placeholder` to show some text when the field is empty.

To control their state we can use the `value` attribute.

We can use the `onInput` attribute to receive a message with the current value in the input field each time the input changes (i.e. the user types something in that field).

Listing 7.1.1 of `Inputs.elm (accountDetailsView)` Elm code

```elm
accountDetailsView : Model -> Html Msg
accountDetailsView { username, password } =
    let
        inputAttrs ty p v msg =
            [ type_ ty, placeholder p, value v, onInput msg ]
    in
    div []
        [ input (inputAttrs "text" "username" username UsernameChanged) []
        , input (inputAttrs "password" "password" password PasswordChanged) []
        ]
```

### 7.1.2 Checkboxes

Checkboxes as simple you as expect, they can be either checked or not. Usually checkboxes are used to select zero or more items from a set, for example extra toppings for a pizza.

Checkboxes are also created with the `input` tag, by setting its `type_` attribute to `checkbox`.

To control their state, we can use the `checked` attribute.

We can use the `onCheck` attribute to receive a message with the current state of the checkbox each time the user clicks on it to check or uncheck it.

Listing 7.1.2 of `Inputs.elm (activateAccountView)` Elm code

```elm
activateAccountView : Bool -> Html Msg
activateAccountView yes =
    div []
        [ input [ type_ "checkbox", onCheck SetActivateAccount, checked yes ] []
        , text "Activate account?"
        ]
```

### 7.1.3 Dropdown lists (select)

Dropdown lists provide a list of options, and one option must always be selected (by definition). This is usually used to select from a set of disjoint options, for example the size or color of a shirt.

Dropdown lists are created using the `select` tag and `option` tags. The dropdown list will only display the options that are inside the `option` tag. Each `option` tag has an `value` attribute that **must** be set, because that value will be sent sent when the selection changes. This allows us to have arbitrary text to be displayed for each option and use simpler and shorter names for our internal logic.

To control their state we can use the `selected` attribute (not used in this example).

We can use `onInput` attribute to receive a message with the currently selected option each time the user changes the selection. Note that the message will contain a `String` which will need to be converted to our desired format (likely the variants of a sum type) in the `update` function.

Listing 7.1.3 of `Inputs.elm (accountTypeView)` Elm code

```elm
accountTypeView : Html Msg
accountTypeView =
    div []
        [ select [ Html.Events.onInput SelectedValue ]
            [ option [ value "User" ] [ text "User" ]
            , option [ value "Admin" ] [ text "Admin" ]
            ]
        ]
```

Exercise 7.1.1 *

Add a button with the text `Create account` that is disabled if the password or username fields are empty. You don't have to add the `onClick` attribute to the button.

Exercise 7.1.2 **

Add a new input field where the user has to repeat their password, to ensure that the password matches. Display a green success message if the passwords match or red error message if the passwords are different. The submit button should also be disabled if the passwords don't match.

Exercise 7.1.3 **

If the account is for administrators (has type `Admin`), the password must have at least 12 characters. For regular users it must have at least 8 characters.
Display an error message and disable the submit button if the password doesn't meet the specified criteria.

## 7.2 HTTP and REST: Basic concepts and terminology

### 7.2.1 HTTP

HTTP is the core communication protocol used by web applications. It is a request-response based, stateless protocol built over TCP.

For this lab all you need to know is:

- During the data exchange the client sends a *request* and the server sends a *response*.
- HTTP has *verbs* (methods) that are used to achieve various actions:
  - `GET` - retrieve data from the server
  - `POST` - send data to the server
  - Others that we won't use: `HEAD`, `PUT`, `PATCH`, `DELETE`, etc.
- A URL (Uniform Resource Locator) is the address of the server:
  - Its simplified format is: `protocol://domain/path?parameters`

$$\underbrace{\texttt{https}}_{\text{protocol}}\texttt{://}\underbrace{\texttt{api.github.com}}_{\text{domain}}\underbrace{\texttt{/users/rtfeldman/repos}}_{\text{path}}\texttt{?}\underbrace{\texttt{sort=pushed\&direction=desc}}_{\text{parameters}}$$

- A request consists of a request line, a series of headers and optionally a body:
  - The request line is always present and contains the method, URL and HTTP version
  - The headers are used to control various aspects of the data transfer from the client side
  - The body is the data sent by the client. It is not present in `GET`, but is present in `POST`
- A response consists of a status, a series of headers and a body:
  - The status is a number, that represents the status of the response. For example 200 means success
  - The headers are used to control various aspects of the data transfer from the server side
  - The body is the data returned by the server

### 7.2.2 REST

REST (Representational State Transfer) is an architectural style (i.e. specification) for building APIs for web services using HTTP. Such an API allows *clients* to access *resources* on the *server*. Services that conform to the REST specification are called RESTful services.

REST defines a set of constraints, of which the following are of interest to us:

- Each request must be stateless and fully independent
- Each resource must be uniquely identifiable

### 7.2.3 Exploring a REST API

The API we are going to use today is `https://restcountries.com/`, which provides data about countries.

The simplest way to get started is to visit `https://restcountries.com/v3.1/all` in your browser. You should obtain a page that contains a JSON array that contains each country. This output might not be very easily readable, so we can use an app like `https://www.postman.com/` to interact with a REST API.

## 7.3 HTTP requests in Elm by example - Countries app

This app uses `https://restcountries.com/` to displays data about countries. It uses the `http` and `json` packages to fetch data and decode it.

### 7.3.1 Model

For this application we will define our application as a state machine: It will have a set of state that it will go through during the interaction with the user:

Listing 7.3.1 of `Countries.elm (Model)` `Elm code`

```elm
type Model
    = Initial
    | RequestSent
    | Success (List Country)
    | Error Http.Error
```

Our app goes through the following states:

- `Initial` - We have no data to display, so we display the button that will start the HTTP request.
- `RequestSent` - When the user clicks the button, we will transition to this state and stay here while the Elm runtime handles the request.
- `Success` - If the request succeeded, we will transition to this state and display the data that we requested.
- `Error` - If something went wrong during the request, we will transition to this state and display an error message.

### 7.3.2 Msg

Listing 7.3.2 of `Countries.elm (Msg)` `Elm code`

```elm
type Msg
    = GetCountries
    | GotCountries (Result Http.Error (List Country))
```

The `Msg` type will represent the transitions our app can take:

- `GetCountries` - Send a message to the Elm runtime to perform the HTTP request.
- `GotCountries` - The Elm runtime completed our request (successfully or unsuccessfully).

### 7.3.3 Update

Finally the `update` function will implement the transitions of the state machine:

Listing 7.3.3 of `Countries.elm (update)` `Elm code`

```elm
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        GetCountries ->
            ( RequestSent
            , getCountries
            )

        GotCountries (Ok countries) ->
            ( Success countries
            , Cmd.none
            )

        GotCountries (Err err) ->
            ( Error err
            , Cmd.none
            )
```

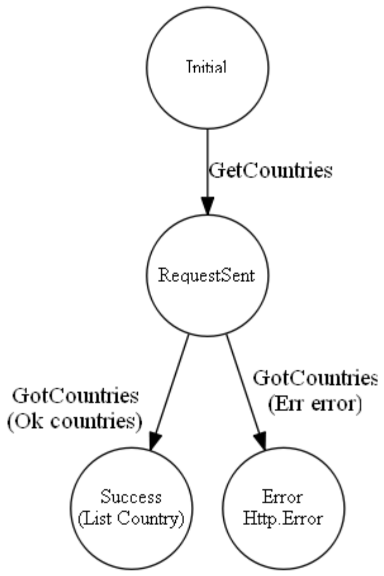

Figure 7.1: State diagram for our app

### 7.3.4 Creating a HTTP request

To perform a HTTP request in Elm, we'll need to ask the Elm runtime to perform it for us, by giving it a command. To create such a command, we can use the `Http.get` function, which expects a record containing 2 fields:

- `url` - The URL where to send the request
- `expect` - What kind response are we expecting. Most of the time, we'll use `Http.expectJson`.

Listing 7.3.4 of `Countries.elm` `(getCountries)` — `Elm code`

```elm
getCountries : Cmd Msg
getCountries = Http.get
    { url = "https://restcountries.com/v3.1/all"
    , expect = Http.expectJson GotCountries (Dec.list decodeCountry)
    }
```

label=ch7:code:getCountries

As we've seen with the generation of random numbers, when the HTTP request is completed, it will produce a message that will trigger un update. However unlike the random number generators, HTTP requests can fail, so the message will actually contain a `Result` that indicates if the request succeeded or failed.

If the request succeeded, we need to take the returned data and see if it matches our expectations, so we can use it. This aspect is specified with the `Http.expectJson` function, which in this case specifies that the response should be in JSON format and should be decoded with the given JSON decoder (which might fail if, for example, some fields are missing).

## Decoding JSON

The Json library offers various functions for dealing with JSON in our applications. It includes 2 modules, the `Encode` module, for creating encoders that are used to convert Elm values to JSON and the `Decode` module, to create decoders that convert JSON values to Elm values.

We will focus only on the `Decode` module, the main functions that we will use are:

- `int : Decoder Int` and `float : Decoder Float` - decodes an int or a float, respectively
- `string : Decoder String` - decodes as string value
- `list : Decoder a -> Decoder (List a)` - decode a list of values
- `field : String -> Decoder a -> Decoder a` - decode a field from an object
- `at : List String -> Decoder a -> Decoder a` - decode a nested field from an object
- `map : (a -> b) -> Decoder a -> Decoder b` - transform a decoder
- `map2 : (a -> b -> c) -> Decoder a -> Decoder b -> Decoder c` - transform multiple decoders

Note how some functions (like `int` and `string`) take no parameters (i.e. behave like constants) and other functions (like `list`, `field` and `map`) take *existing* decoders and return a new decoder.

This is a called a *composable* API, because it allows us to create more complex "logic" from simple building blocks.

In the example below, we use the `map3` function to extract 3 fields from an object, the first being a nested string field, the second a float, and the third a string, then apply the `Country` constructor to the extracted fields.

Listing 7.3.5 of `Countries.elm` `(decodeCountry)` Elm code

```elm
decodeCountry : Dec.Decoder Country
decodeCountry =
    Dec.map3 Country
        (Dec.at  ["name", "common"] Dec.string)
        (Dec.field "area" Dec.float)
        (Dec.field "region" Dec.string)
```

Note that the return type of this function is `Decoder Country`, which means that we can provide it as a parameter to functions like `list`, `field` and `map`, just like the basic functions (`string`, `int`) defined in the library, which is exactly what we do in ?? to decode a list of countries.

### 7.3.5 View

The `view` is again constructed *hierarchically*: the `view` function performs pattern matching on the model and delegates each non-trivial case to a different function.

Listing 7.3.6 of `Countries.elm` `(view)` Elm code

```elm
view : Model -> Html Msg
view model =
    case model of
        Initial ->
            viewInitial

        RequestSent ->
            div [] [ text "Loading..." ]

        Success countries ->
            viewSuccess countries

        Error err ->
            viewError err
```

Note that each function that we delegate to takes a more specific input (i.e. not `Model`), usually the data that was in the matched case (i.e. for `Success countries`, `viewSuccess` takes `List Country`).

Listing 7.3.7 of `Countries.elm` `(viewSuccess)` Elm code

```elm
viewSuccess : List Country -> Html msg
viewSuccess countries =
    div [] ((h2 [] [ text "ok" ])::List.map viewCountry countries)
```

Listing 7.3.8 of `Countries.elm` `(viewCountry)` Elm code

```elm
viewCountry : Country -> Html msg
viewCountry {name, area, region} =
    div [style "border" "solid 1px", style "margin" "2px"]
        [ p [] [text <| "Name:" ++ name]
        , p [] [text <| "Area: " ++ String.fromFloat area]
        ]
```

## 7.4 Documenting functions and checking examples

To write documentation for functions and types you can use block comments, which start with `{-`, end with `-}`, can span multiple lines and support markdown.

We can also include basic usage examples in documentation. The examples are indented with exactly 4 spaces and the expression and result are separated by `-->`:

Listing 7.4.1: The tails function with documentation — Elm code

```elm
{-| Returns all the ends (tails) of a list.

    tails [1, 2, 3] --> [[1, 2, 3], [2, 3], [3], []]
-}
tails : List a -> List (List a)
tails l =
    case l of
        [] -> []
        x::xs -> (x::xs)::tails xs
```

To test these examples, we can use the `elm-verify-examples` npm package:

```powershell
powershell session

PS > npm install --global elm-verify-examples
```

To set up these test, after running `npx elm-test init`, we also need to run in the project folder:

```powershell
powershell session

PS > npx elm-verify-examples init
```

This will create a file named `elm-verify-examples.json` in the `tests` folder. Each module that contains documentation with examples needs to be added to the list under the `tests` key this file.

Listing 7.4.2: The elm-verify-examples.json file — Elm code

```json
{
    "root": "../src",
    "tests": ["Documentation"]
}
```

Then, to generates tests for the examples:

```powershell
powershell session

PS > npx elm-verify-examples
```

Finally, to run the test we use the usual:

```powershell
powershell session

PS > nxp elm-test
```

Which can be written as one command:

```
                                                    powershell session
PS > npx elm-verify-examples; nxp elm-test
```

## 7.5 Compiling your application to HTML

To create HTML file from your project, simply run **`elm make`** giving it the file that contains the `main` function for application that you want to build. For this lab, we will use:

```
                                                    powershell session
PS > elm make src/Countries.elm
```

You should see a file named **`index.html`** that you can open with your browser to see your app.

## 7.6 Practice problems

**Exercise 7.6.1** *

Run the `tails` function to check if it behaves according to the examples. If not, find and fix the bug.

**Exercise 7.6.2** **

Implement the `combinations` function such that the tests pass.

**Exercise 7.6.3** *

Display the population and population density (population/area) for each country.

**Exercise 7.6.4** *

Sort the countries by area in descending order.

**Exercise 7.6.5** *

Add a checkbox which allows the user to change the current sort order of countries to ascending (i.e. if the checkbox is is unchecked the sort order will be descending, if it checked the sort order will be ascending).

**Exercise 7.6.6** **

Add a text field which will allow to filter countries by name, using the `String.contains` function.

**Exercise 7.6.7** `optional`

Add a dropdown list (select element) to select the field for the sorting. The options should be: population, area, population density.

**Exercise 7.6.8** `optional`

Add a set of checkboxes to allow filtering the currently shown countries by region.

Hints:

- Do you know all options in advance (i.e. all the values that the region field can take)?
- If yes, is it better to hardcode these options, or is it better to obtain them dynamically?
- What data structure will you use to store the selected regions in each case (hardcoded or dynamic)?