Assignment 1

Elm project - Hacker News client

Deadline: Monday, November 24th, 2025, 23:45
Late submissions policy: 5 (out of 30) points penalty for each late day

1.1 Submission instructions

1. Unzip the `Elm-Project.zip` folder. You should find (among others):
   - `src` folder - your workspace
   - `tests` folder - self evaluation tests
   - `scripts` folder - utility scripts
   - `.gitignore` - if you want to use version control
   - `elm.json` - elm project configuration
   - `package.json` - npm project configuration
2. Run `npm install` to install the dependencies needed for the automated tests.
3. Edit the source files in the `src` folder with your solutions.
4. When done, run `npm run zip` which will create a zip archive with the `src` folder.

In order to be accepted, your solution:
- **must compile successfully**
- **must not modify the provided elm.json file in any way (i.e. you are not allowed add any new dependencies)**

1.2 Project resources

Table 1.1: Project Resources

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

Table 1.2: Extra Resources - Talks about how to design Elm apps

Resource	Link
The life of a file - Evan Czaplicki	`https://youtu.be/XpDsk374LDE`
Making Impossible States Impossible - Richard Feldman	`https://youtu.be/IcgmSRJHu_8`
Immutable Relational Data - Richard Feldman	`https://youtu.be/28OdemxhfbU`
Make Data Structures - Richard Feldman	`https://youtu.be/x1FU3e0sT1I`

1.3 Project description, goals and non-goals

In this project you will develop basic Hacker News client. It will fetch the top stories and show them in a table giving the user the option to filter and sort posts by various criteria.

The main goal of the project is to get hands-on experience for building a close to real-world app, that displays useful data, can retrieve data from a server and has a decent test suite to ensure that it works properly.

There are also non-goals for this project, the main one being styling (i.e. the look of the final application) - don't spend time on styling before the logic of the app is complete. Other non-goals include handling and validating more complex inputs from the user - while this use case certainly appears in the real world, it is often quite tedious and time consuming to implement and thus it is better to spend more time on simpler features that can still make a useful app.

1.4 Grading

This project is worth 30% of your final lab grade. You can obtain in total 30 points:

- 10% (3 points) are awarded by default (i.e. represent the starting grade)
- 50% (15 points) come from public tests (i.e. that you can run to check your implementation)
- 20% (6 points) come from manual grading or hidden tests (i.e that are not available to you, but will be run when grading your project)

- 20% (6 points) come from coding style (graded manually, see section 1.7)

The tests will cover all functional requirements, but you can implement as much or as little as you consider adequate. The grade for functional requirements will be calculated from the number of tests that pass (failing tests most likely mean that a requirement is missing or is not implemented correctly).

1.5 Getting started with the development

Starting code

Most of the logic in the `Main.elm` and `Model.elm` files is already implemented: `Main` contains the basic skeleton for the app and `Model` contains the data definitions for the model. The other files under the `Model` folder also contain some functions that are already implemented.

It is highly recommended that you spend some time to understand the existing code before starting to write your solutions.

Development process

After you unzip the project files, you have run `npm install` to install the dependencies. This has to be done only once.

Then, you should run `npm test` to confirm that the tests fail because of `Debug.todo`. It might help to replace `Debug.todo` with a concrete value that makes the function compile, just to see that all the tests fail. Such values are placed as a comment just below the first line of the function. See section 1.8 for more details about testing.

Finally, you should start `elm reactor`, open the `src/Main.elm` file (both in reactor and in your editor) and start by commenting most of the `Main.view` function to focus on getting the other `view`s to compile. After the view you're working on compiles you can run tests to see if they pass. Then you can slowly uncomment functions in `Main.view` to repeat this procedure.

If you would like to work offline, you can use the `src/Reactor.elm` file and the `npm run server` command to use a local server to obtain some hardcoded data that matches the shape of the data returned by the real API.

1.6 Project tasks (functional requirements)

Exercise 1.6.1 `3p (public) + 1p (hidden)`

Implement the functions in the `Cursor` module according to the documentation comments and examples.

Grading:

0.75p	`forward`
0.75p	`back`
0.5p	`fromList`
0.5p	`toList`
0.5p	`length`

Exercise 1.6.2 3p (public)

Complete the `View.Posts.postsTable` function to show a table that contains the score, title, type, posted date and link of each `Post`.

Grading:

- 1p For showing a header with one column for each field: score, title, type, posted date and link.
- 2p For showing a row for each post:
  - 1p Each cell in the row should show the corresponding field of the post from the header (i.e. show the score of each post in the "score" column). You can format the `time` field by using `Util.Time.formatTime Time.utc post.time`.
  - 1p Each cell should have a class with the name `post-<field_name>`:
    - ∗ the `score` field should have `class` *post-score*
    - ∗ the `title` field should have `class` *post-title*
    - ∗ the `url` field should have `class` *post-url*
    - ∗ the `type` field should have `class` *post-type*
    - ∗ the `time` field should have `class` *post-time*

Exercise 1.6.3 1.5p (public)

Complete the `View.Posts.postsConfigView` function such that it displays the configuration options related to the posts table.

Grading:

- 0.5p For showing an input where the user can select the number of posts to show (10, 25 or 50)
  - – this should be implemented as a `select` element, with `id` *select-posts-per-page*
- 0.5p For showing an input where the user can select the field by which to sort the posts table (score, title, date posted, unsorted)
  - – this should be implemented as a `select` element with `id` *select-sort-by*
- 0.25p For showing an input where the user can choose whether job posts are shown
  - – this should be implemented as a checkbox with `id` *checkbox-show-job-posts*
- 0.25p For showing an input where the user can choose whether posts without an url are shown
  - – this should be implemented as a checkbox with `id` *checkbox-show-text-only-posts*

Exercise 1.6.4 1.5p (public) + 1p (hidden)

Complete the `Model.PostConfig.filterPosts` function that takes all the loaded posts and returns a subset according to the provided `PostsConfig`.

Grading:

- 0.25p Text only posts are filtered from the list if the `showTextOnly` is `False`
- 0.25p Job posts are filtered from the list if the `showJobs` is `False`
- 0.5p The final list includes at most `postsToShow` posts
- 0.5p The final list is sorted according to `sortBy`

Exercise 1.6.5 — `1.5p (public) + 1p (hidden)`

Complete the functions in `Model.Post`, `Model.PostIds` and `Model.PostsConfig`.

Grading:

0.5p	`PostIds.decode`
0.5p	`Post.decode`
0.25p	`PostIds.advance`
0.25p	`PostsConfig.sortFromString`

Exercise 1.6.6 — `1.5p (public)`

Complete the `Util.Time.formatDuration` and `Util.Time.durationBetween` functions. Then update the `View.Posts.postsTable` function to display the duration since the post was submitted.

Grading:

0.5p	`Util.Time.formatDuration`
0.5p	`Util.Time.durationBetween`
0.5p	Add between parentheses to the cell where the submission time is displayed, such that it displays the time passed (i.e. relative duration) since the post was submitted (i.e. instead of displaying just the absolute date like “Nov 1, 2025”, you should display “Nov 1, 2025 (1 day and 2 hours ago)”).

Exercise 1.6.7 — `3p (public) + 3p (hidden)`

Complete the functions in `Main` and `View.Posts` to update the model when the post table configuration changes
After completing this exercise, when you change one of the options (for example the number of posts to show) the change should be reflected in the posts table (i.e. the number of displayed posts should change).

Grading:

1p	(manually graded) Complete the `Model.PostsConfig.Change` type to hold the data for each possible change
1p	Implement `Model.PostsConfig.applyChanges` to return the updated configuration based on the current configuration and a `Change`
1p	Update `View.Posts.postsConfigView` to send messages for each change
1p	Complete `Main.update` to update the configuration using `Model.PostsConfig.applyChanges`
2p	Extra hidden tests

1.7 Coding style (non-functional requirements)

Exercise 1.7.1 — 3p

Properly use Elm language features and library functions. Examples include:

- Lambda functions
- Pipelines and function composition
- Functions found in the standard library:
  - Functions for list processing (`List.map`, `List.filter`, `List.foldl`, etc.)
  - Functions for error handling (`Maybe.map`, `Maybe.withDefault`, etc.)

Note that the goal of the list above is only to give you a general idea of the features that you should consider when writing the code. Your goal is to show that you know when to use and when to not use various features. For example, there are two extremes that you should clearly avoid:

- writing Elm code that tries to mimic an imperative style (i.e C or Java) or is not using any Elm language features
- (ab)using all Elm features in a way that makes the code harder to understand (obfuscates the intent)

Exercise 1.7.2 — 3p

Use a proper coding style:

1.5p Descriptive names for data definitions and functions
1.5p Readable code structure (proper use of indentation, reasonably sized functions)

1.8 Testing your implementation

The project contains both unit test that can be run with `elm-test` and examples that can be run with `elm-verify-examples`. You have to run `npm install` (once) to run tests.

To run all test and see your final grade, use:

```
powershell session
PS > npm run grade
```

To run all test and see detailed explanations for all failures, use:

```
powershell session
PS > npm test
```

To run all tests manually, you can use:

```
powershell session
PS > npx elm-test
```

If the tests fail with an exception containing `Error: TODO in module`, you have to replace the respective call to `Debug.todo` with a normal Elm value that makes the code compile.