Elm Friendly functional frontend

Model all state → `view` → **View** Html Msg → `user event` → **Msg** what happened → `update` → **Update** returns new Model

The Elm Architecture

CONCEPTS

Read compiler errors top to bottom: they usually name the fix.

Install the tools with npm: npm install -g elm elm-format elm-test.

PROJECT

`elm init`
Create elm.json and src/

`elm install <author/package>`
Add a dependency

`elm repl`
Start an interactive session

BUILD

`elm make src/Main.elm`
Compile to index.html

`elm make src/Main.elm --output=elm.js`
Compile to a JS file

`elm make src/Main.elm --optimize`
Optimized build, no Debug

`elm make src/Main.elm --debug`
Add the time-travel debugger

BUILD (cont.)

`elm make src/Main.elm --output=/dev/null`
Type-check only, no output

`elm reactor`
Serve the project at :8000

`elm reactor --port=<n>`
Serve on another port

PACKAGES

`elm publish`
Publish a package version

`elm bump`
Compute the next version

`elm diff`
API changes since last publish

`elm install elm/json`
JSON encode and decode

`elm install elm/http`
HTTP requests

`elm install elm/browser`
Browser.element and Browser.application

TOOLING

`elm-format src/ --yes`
Format all files in place

`elm-format src/ --validate`
Fail if not formatted (CI)

TOOLING (cont.)

`elm-test init`
Create tests/ and test deps

`elm-test`
Run all tests in tests/

`elm-test <file>`
Run one test file

`elm-test --watch`
Re-run on change

`elm-test --fuzz <n>`
Fuzz runs per test

SYNTAX

`module Main exposing (main)`
Module header

`import Html exposing (Html, div, text)`
Import names

`import Html.Attributes as A`
Import with alias

`name : Int → String`
Type annotation

`type alias User = { name : String }`
Name a record type

`type Msg = Inc | Dec`
Custom type

`case x of ... → ...`
Pattern match

SYNTAX (cont.)

`if a then b else c`
Conditional, always an expression

`let y = 1 in y + 1`
Local bindings

`\x → x + 1`
Anonymous function

`{ model | count = 1 }`
Update record fields

`model.count / .count`
Field access / accessor function

`x |> f |> g`
Pipe left to right

CORE LIBRARY

`List.map f xs`
Transform each item

`List.filter f xs`
Keep matching items

`List.foldl f init xs`
Reduce a list

`String.fromInt n / String.toInt s`
Int to string / parse (Maybe)

`Maybe.withDefault 0 m`
Unwrap Maybe with fallback

`Maybe.map f m`
Apply inside Maybe

CORE LIBRARY (cont.)

`Result.map f r`
Apply inside Result

`Dict.fromList pairs`
Build a Dict

`Dict.get key d`
Lookup, returns Maybe

`Debug.log "label" value`
Print and return value

PROGRAMS

`Browser.sandbox`
Simple app: init, view, update

`Browser.element`
App with Cmd, Sub and flags

`Browser.application`
Full app with URL routing

`Cmd.none / Sub.none`
No effect / no subscription

`Http.get { url = u, expect = e }`
GET request as a Cmd

`Decode.field "name" Decode.string`
JSON decoder for a field

`Html.Events.onClick Msg`
Send a Msg on click

`Html.Events.onInput Msg`
Send a Msg on typing