LifeScale – Developer's Guide:

LifeScale is an application that helps users to balance certain elements (such as food intake, work/life, etc) of their lives. See the user guide for more explanation of how this works.

It is advised that you read the user guide, before reading through this.

Contents:

Technology Used:

At this time, this is a front-end concept application. Therefore, the back-end has been mocked with JSON-Server.

The front-end was developed with React, TypeScript, and Sass.

Standard Styles:

The standard styles can be found in */front_end/src/stdStyles*

As Sass is used to pre-process the application's CSS, there are some standard variables/mixins that you should utilise, to keep the design of the application's elements consistent. These are listed below:

stdFonts.scss

- **stdPrimaryFont** – This is the main font for the application.
- **stdHeavyFont** – This is a heavier font, for use if required in headings, etc.

stdColors.scss

- **$stdPrimaryTxtColor** – This is primary text color for the application
- **$stdSecondaryTxtColor** – This is secondary text color for the application
- **$stdPrimaryBackColor** – This is primary background color for the application
- **$stdSecondaryBackColor** – This is secondary background color for the application
- **$stdPrimaryBorderColor** – This is primary border color for the application
- **$stdSecondaryBorderColor** – This is secondary border color for the application

stdBorders.scss

- **stdBorder** (parameters are $color, $weight) – This is a mixin that provides a customisable border, consistent with the other borders in the application. You can pass a color and weight to this.
- **stdPrimaryBorder** – This is a mixin that provides the standard border for the application
- **stdSecondaryBorder** – This is a mixin that provides a standard secondary border, for use on secondary elements in the design.

stdButtons.scss

- **stdPrimaryButton** – This is a mixin that provides the standard button styles

stdCard.scss

- **stdCard** – There are components, known as "cards", that are designed to represent an item in a list that is displayed within a flex container. This mixin provides the standard styling for cards. (See the CardDisplay, EditableItemCard, and AddItemCard components)
- **stdCardSingleCharacterSpan** – Some cards will contain just a single character, within a span (See the AddItemCard component, as that does this with a "+" character). This mixin provides the standard styling for the span.

stdForms.scss

- **stdForm** – This is a mixin that provides the standard styling for forms.
- **stdInnerForm** – This is a mixin that provides the standard styling for scondary/inner forms.

stdPercentageStatistic.scss

- **stdPercentageStatistic** – There are sometimes statistics displayed in the system, that have a label and a percentage value. This is the standard styling for those statistics.

stdResponsiveConfig.scss

- **$smallScreenWidthPixels** – This can be used in media queries, to determine what width is considered small
- **stdResponsiveUserTextConfig** – This is a mixin which provides styling to text, to make it extra responsive.

Icons:

Icons can be found at */front_end/src/icons*

These are icons/SVGs that can be used within the system.

SVG:

- **list.svg** – This icon is used to express that the element can be expanded (such as a drop down for the navigation bar, when on mobile devices)

Interfaces:

Interfaces can be found at */front_end/src/interfaces*

There are multiple TypeScript interfaces used in this application. These can be seperated into the following categories:

- **Data** – Represents data that can be persisted/retrieved from the API
- **API Access** – Represents objects that can be used to provide access to the API
- **UI** – Represents a collection of properties that are required by certain components. Used to keep the things clean when passing this data through multiple components.

Data Interfaces:

- **IUser** – Represents a user object.
- **IScale** – Represents a "scale" object. "usesTimespans" is a boolean that tells the application if that scale should give the option to set the minute count when creating timespans (if not, the minute count should default to 1). "displayDayCount" is the amount of days worth of timespans that should be taken into account when calculating timespan statistics, to be displayed to the user.
- **ICategory** – Represents a "category" of events that can be recorded. "desiredWeight" is the weight (on the displayed scale) that the user is aiming to have for this category (1 is standard, but can be more or less to give the category more/less space on the desired scale).
- **IAction** – Represents a type of event that can be recorded. "weight" is the weight that timespans for this action should get on the scale (Standard weight is 1. The weight is used when calculating the balance of each category's timespan's values).
- **ITimespan** – Represents an actual recorded event. The default value for "minuteCount" should be 1.

API Access Interfaces:

- **IUserService** -- A "service" class that implements this interface will be used to login, load, access, and set a user's information. See the comments against this interface for full information.

UI Interfaces:

- **IActionFormItem** – Used to represent a single action's required form info/callbacks.
- **IActionHistoryItem** – Used to represent a historical occurence of an action (as in, a timespan).
- **ICategoryColorData** – Used by the CategoryColorProvider utility class to represent a information on a catgeory color (See information on CategoryColorProvider, for full information.).
- **ICategoryFormItem** – Used to represent a category's required form info/callbacks.
- **IPercentageStatistic** – Used to represent a percentage statistic's information (such as label and percentage). A percentage statistic can have a further breakdown of the data it represents, so the "children" property can store inner statistics.
- **IRecordedActionFormItem** – Used to represent a recorded action's form data/setStates/callbacks.
- **IScaleBalanceItem** – ScaleBalanceItems represent an item (say, a category) to be placed on

a scale display.

- **IScaleFormItem** – Used to represent a scale's required form info/callbacks.
- **IScaleLink** – Used to define a link to a scale's details, in the navigation area

User Services:

User services can be found at */front_end/src/userServices*

Whenever the application needs to access the API, or the user's data, it should do this through an implementation of the IUserService interface. Classes that implement this interface will control the access to the API, so the rest of the application is de-coupled from the API, and therefore free from concern if the API details need to change.

- **MockJSONServerUserService** – This is a UserService implementation designed to interact with a JSON-Server mock back end API.

  As this is interacting with a mock API, certain functionality that would be in a real API (such as validating logins, etc) will be handled by this instead (this is purely designed for mocking a real back-end)
- **TestingDummyUserService** – This implementation of IUserService was created for use in tests. Tests can create an instance of this, then overwrite the required methods with mocks.

Utility Classes:

Utility classes can be found at */front_end/src/utility_classes*

CategoryColorProvider:

The values stored in the database for ICategory colors will need to be consistent (though more can be added, without issue), however the actual color values themselves will need to be able to change, based on design needs.

Therefore, an instance of this class will provide the "real values" for a given color name (or vice-versa). This means the stored color name (the consistent data over time) can be converted back and forth to the potentially changing CSS colors that are actually displayed

Utility Functions:

Utility functions can be found at */front_end/src/utility_functions*

convertDateToInputString:

Used to convert a Date object into the string representation required in a "date" type input element (as these inputs require a specific date format).

Components:

Components can be found at */front_end/src/components*

There are 3 kinds of React components in this application: display-level components, utility components, and logic-container components.

The display-level components have very little (if not zero) business logic. They exist to display and set state.

The utility components are usually wrappers for other components, that handle a certain operation (such as displaying "loading" placeholders, or positioning components in a responsive way).

The logic-container components are wrapper components, that handle the logic/state/api access/etc for components.

If you see that a component has a logic container component in the same directory, you should use that instead of the display component.

Entry Point:

- **App** – Entry point for application (that being said, this needs to be wrapped in a router component, in the index.tsx file).

  This will handle the display of components, based on the current route. It will also handle the redirecting when a user is at a route they shouldn't be on, based on their auth status.

Logic-Container Components:

- **ActionsFormLogicContainer** – Wrapper component that controls the business logic for the ActionsForm component. See the ActionsForm component for more description.
- **AmendActionHistoryPageLogicContainer** – Wrapper component that controls the business logic for the AmendActionHistoryPage component. See the AmendActionHistoryPage component for more description.
- **CategoryDetailsFormLogicContainer** – Wrapper component that controls the business logic for the CategoryDetailsForm component. See the CategoryDetailsForm component for more description.
- **ChangePasswordFormLogicContainer** – Wrapper component that controls the business logic for the ChangePasswordForm component. See the ChangePasswordForm component for more description.
- **LoginPageLogicContainer** – Wrapper component that controls the business logic for the LoginPage component. See the LoginPage component for more description.
- **RecordActionFormLogicContainer** – Wrapper component that controls the business logic for the RecordActionForm component. See the RecordActionForm component for more description.
- **RequestPasswordPageLogicContainer** – Wrapper component that controls the business logic for the RequestPasswordPage component. See the RequestPasswordPage component for more description.
- **ScaleDetailsFormLogicContainer** – Wrapper component that controls the business logic for

the ScaleDetailsForm component. See the ScaleDetailsForm component for more description.

- **UserDetailsFormLogicContainer** – Wrapper component that controls the business logic for the UserDetailsForm component. See the UserDetailsForm component for more description.
- **UserHomeScreenLogicContainer** – Wrapper component that controls the business logic for the UserHomeScreen component. See the UserHomeScreen component for more description.
- **UserNavBarLogicContainer** – Wrapper component that controls the business logic for the UserNavBar component. See the UserNavBar component for more description.

Utility Components:

- **DropdownContentBar** – Displays a bar that will expand, as a drop-down, to display its children.
- **LoadedContentWrapper** – Used when you want to render something, but may not have all the data needed to render it yet.

  This will render a loading message or icon (this may change thoughout the life of the project), if nothing is provided in the "render" prop.

  If the errorMessage prop is provided, that will be rendered instead.
- **NavigatableContentWrapper** – Used when you want to render content that will be navigatable, and responsive. You provide the navigation bar in the navigationBar prop, and then provide the rest of the content as children. This component will then render these, in a responsive layout.

Display Components:

- **ActionHistoryItem** – Used to display an occurrence of an action.
- **ActionsForm** – Used to Create/Update/Delete actions.
- **AddItemCard** – Card to be an item in a flex container. This card will be used to take you to a screen to add a new item
- **AmendActionHistoryPage** – Used to delete previous occurences of actions on a given scale, and to add new ones.
- **CardDisplay** – Acts as a flex container, to display its children.
- **CategoryDetailsForm** – This is used to create and edit categories in the system. The container component should handle the logic.
- **CategoryDetailsFormPartial** – Used to display and set basic category details (not including setting actions).
- **ChangePasswordForm** – Used to allow a user to change their password (and uses a boolean prop to set if the new password has been confirmed).
- **EditableItemCard** – Card to be an item in a flex container. This card will be used to display items that can be edited on another screen (the Edit button will take you to the other screen -- via the editCallback prop).
- **EmptyContentMessage** – Used to display a message, when the container would usually display content, but none is available (e.g. If no scales have been created, this can be displayed instead)
- **ErrorMessageDisplay** – Used to display critical error messages.
- **LoginForm** – Display-layer component to display a form for users to log in

- **LoginPage** – Used to login to the application (also to provide links to register an account, and for when users forget their passwords).
- **PasswordFormPartial** – This is a form partial for password setting/resetting.

  If the password and confirmation password don't match, it will run the setPasswordIsConfirmed prop with a value of false. Otherwise, a value of true.

  This will display a BadSaveMessage if the passwords don't match

  The passwordLabel prop will determine what the form labels say (maybe you want it to say "New Password", instead of just "Password"). The confirmation label will always start with "Confirm". The default passwordLabel is "Password".
- **PercentageStatistic** – Used to display percentage information for actions/categories/etc that the user has recorded (say, what percentage of events on a scale are of the action type in question).
- **RecordActionForm** – Used to record an occurrence of an action.
- **RequestPasswordForm** – Display layer component to display a form for users to request a new password.
- **RequestPasswordPage** – Used to let the user request a new password, when they have forgotten their's.
- **SaveMessage** – This is a base class for the good/bad save messages. They should be used instead of this.
- **GoodSaveMessage** (found at */front_end/src/components/SaveMessage/GoodSaveMessage*) – Used to display a message when something has been sucessfully saved.
- **BadSaveMessage** (found at */front_end/src/components/SaveMessage/BadSaveMessage*) – Used to display a message when something has been unsucessful in saving
- **ScaleBalanceDisplay** – Used to display different items (say, categories) on a scale -- items take up more/less visual space. based on how heavy their weight is (so more space for heavier, less for lighter).
- **ScaleDetailsForm** – This is used to create and edit scales in the system (it also displays a CardDisplay, to provide links to create/edit categories). The container component should handle the logic.
- **ScaleDetailsFormPartial** – Used to display and set basic scale details (not including setting categories).
- **ScalePrimaryDisplay** – Used to display the desired balance scale, and the current balance scale. Also provides a button, to be used to take the user to the screen to edit the scale details.
- **ScalesNavList** – Takes an array of IScaleLink objects (defined in this file) and creates a ul of react-router Links.
- **ScaleStatisticDisplay** – Displays statistic percentage data for categories/actions. Also has a button to go to the page to amend the action history for this scale.
- **SingleActionForm** – Used to display and save an action (not a usage of an action, but the details of the action-type itself).
- **TimespanFormPartial** – Used to display and set an amount of time (in minutes). This will also provide the ability to set it in hours (however the component will still pass back up the value in minutes).
- **UserDetailsForm** – This is used to create and edit users in the system. The container should

provide an IUser, with blank or default data (the ID can be whatever the container decides -- probably a blank string).

The container can also enter a password form to be rendered (will differ depending on if it's a creation, or password change)

- **UserDetailsFormPartial** – Used to display and set basic user account details (not including password set/reset).
- **UserHomeScreen** – Displays the home navigation bar, and the information about the selected scale (with a RecordActionForm, to record new timespans).
- **UserNavBar** – Navigation, once logged into the app. This provides links to edit user details, create scales, and view scales.

  If the scales list is still being loaded, the scaleLinks prop should be an empty array.

Testing:

Front-End Testing:

The tests for the front-end use Jest. You can run them, in the *front_end* directory, with the below command:

npm test

Back-End Testing:

As the back-end is currently mocked with JSON-Server, there are no back-end tests at this time.

E2E Testing:

The E2E testing is done with Cypress. To run these tests, you first need to go to the *back_end* directory, and start the JSON-Server with an alternative command:

npm run serve-test

This will run the back-end with the *e2e-test-db.json* file as the database, instead of the *db.json* file. The tests should clear up after themselves, so if you have data in the regular back-end JSON file, then that might get cleared if you were to use that.

Next, you will need to go to the *front_end* directory, and run the below command:

npx cypress open

This will start Cypress, and you can run the spec of your choice.