Contact Person	John Torous, BIDMC Division of Digital Psychiatry
Description	LAMP Platform (frontend/backend)
Reviewer	2muchcoffee team
Date	02/12/2021

Backend

Conclusions

To improve scalability of the project and enable development of new essential functionality, it is imperative that the backend code be refactored and restructured to overcome the fundamental issues it currently faces. For example, not all basic or advanced features of Typescript are used effectively, and there are several cases where functionality that was re-developed specifically for this project could benefit from switching to existing tailor-made solutions and contemporary frameworks.

Suggestions

There are several structural errors that further complicate the already complex code, along with usage of deprecated or no longer recommended approaches to achieve certain functionality. It would be appropriate to refractor and adopt modern approaches and frameworks/tools that solve these issues, such as NestJS. Further recommendations include:

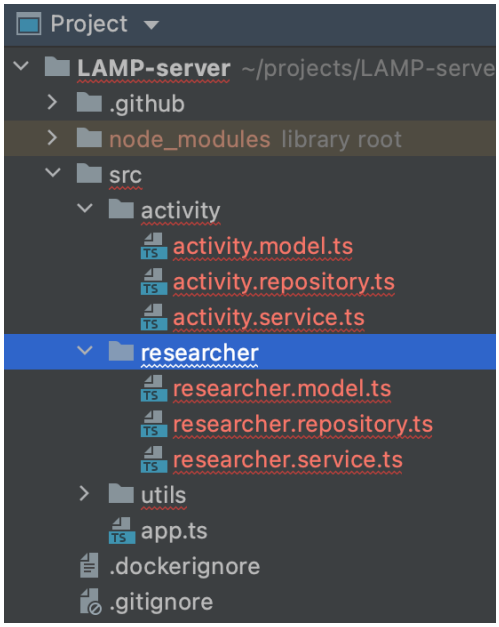
- Moving common/shared functionality into middleware functions, for example, for centralized error processing, authentication and authorization, subscription management, and so on.
- Migrating the codebase to fully strictly typed, by describe all interfaces instead of using the “any” type, which resolve many serious internal errors in the codebase.
- Reducing and removing as many callbacks and nested functions as possible.
- Restructuring the codebase by separating it into modules, with names corresponding to functionality performed by the specific module.
    - This would be a much better approach instead of separating services/controllers/models into groups, as well as the separation of each module between router/controller/service/repository.

Project Dependencies

- There are many third-party packages present that are unused by the codebase (“fastify”, “mssql”, “node-cron,” and so on), as well as several mutually-exclusive dependencies (“fastify” vs. “express”).

Project Structure

- As previously stated, the code structure is not ideally organized to further scale and advance the project given the intended user base and required functionality. It would be logical and suitable to separate the code into catalogues with the corresponding names reflecting the functionality represented:



(An example, which implements the activity and researcher functionality, into separate catalogs. The whole project should be re-formatted in this way).

- The service catalogue contains files in which a controllers' functionality are described. "ListenerAPI" and "PushNotificationAPI," which are located in the utility catalogue, also contain controllers' functionality for API subscriptions and push notifications; this logic is also shared and located in different places across the codebase.
    - In other words, the same duplicated functionality is presented in different places of the app, and resolving a bug may become more complex as the same bug may need to be resolved multiple times, decreasing development productivity.
    - The "PushNotificationAPI" logic is best refactored into a separate microservice module that communicates with the backend, following the a stricter pattern for the delegation of responsibility and decreasing the potential for breakage of one component to affect others.
- Controller functionality is overloaded with logic, which would be important to move into separate services and reduce shared/rewritten/duplicated code.
- The code currently in the service catalogue should be migrated into classes with static functions that are then called by the Express route callbacks.
- There is a large surface of fully commented-out (disabled) code as well as completely unused code that complicates the codebase and will make it difficult to implement new functionality; such code should be entirely removed.
- The possibilities enabled by Express middleware functions are not used; for instance, authentication/authorization should be implemented as a middleware function, potentially using a declarative role-based authentication library like "CASL" instead of the complicated and unclear approach.
- Exceptions are currently manually thrown in the controllers, and should instead be centralized and converted to Express middleware functions.

```
try {
  let researcher_id = req.params.researcher_id
  const researcher = req.body
  researcher_id = await _verify(req.get("Authorization"), type: ["self", "parent"], researcher_id)
  const output = { data: await ResearcherRepository._update(researcher_id, researcher) }
  researcher.action = "update"
  researcher.researcher_id = researcher_id

  //publishing data for researcher update api with token = researcher.{researcher_id}
  PubSubAPIListenerQueue.add({ topic: `researcher.*`, token: `researcher.${researcher_id}`, payload: res
  PubSubAPIListenerQueue.add({ topic: `researcher`, token: `researcher.${researcher_id}`, payload: resea
  res.json(output)
} catch (e) {
  if (e.message === "401.missing-credentials") res.set("WWW-Authenticate", `Basic realm="LAMP" charset="
  res.status( code: parseInt(e.message.split( separator: ".")[0]) || 500).json( body: { error: e.message })
}
```

```
try {
  const researcher = req.body
  const _ = await _verify(req.get("Authorization"), type: [])
  const output = { data: await ResearcherRepository._insert(researcher) }
  researcher.action = "create"

  //publishing data for researcher add api with token = researcher.{_id}
  PubSubAPIListenerQueue.add({ topic: `researcher`, token: `researcher.${output['data']}`, payload: res
  res.json(output)
} catch (e) {
  if (e.message === "401.missing-credentials") res.set("WWW-Authenticate", `Basic realm="LAMP" charset=
  res.status( code: parseInt(e.message.split( separator: ".")[0]) || 500).json( body: { error: e.message })
}
})
```

Typescript

- The typing across the project is quite weak; there are no interfaces to describe the data structure, and “any” is used everywhere, which can lead to serious internal code errors.

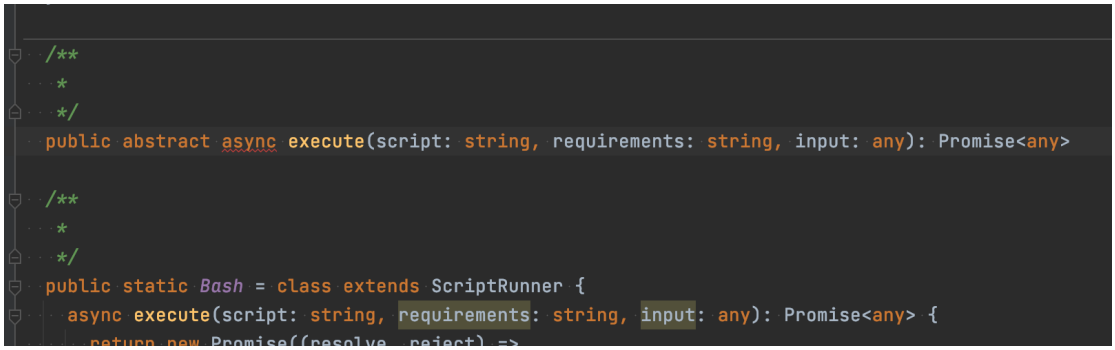
```
PushNotificationQueue.process(async (job: any) => {
  job.data.payload.url = `/participant/${job.data.payload.participant_id}`
  sendNotification(job.data.device_token, job.data.device_type, job.data.payload)
})
```

```
 *
 * @param Participants
 * @param schedule
 */
async function sendToParticipants(Participants: any, schedule: any): Promise<void> {
  Participants = Array.isArray(Participants) ? Participants : [Participants]
  for (const participant of Participants) {
    try {
      const event_data = await SensorEventRepository._select(
```

In the examples presented above, there is no description of the incoming data for functions, and there is no verification whether data is being processed in the functions themselves.

- There is a serious misunderstanding of the role of the “async” keyword across the codebase; for example, for the abstract method the return signature should be “Promise” instead of “Promise<any>” which would allow the concrete subclasses to implement the

function by returning specific concrete types.



Frontend

Conclusions

To improve scalability of the project and enable development of new essential functionality, it is imperative that the frontend code be refactored and restructured to overcome the fundamental issues it currently faces. For example, not all basic or advanced features of Typescript are used effectively, and there are several cases where functionality that was re-developed specifically for this project could benefit from switching to existing tailor-made solutions and contemporary frameworks. Most of the files that comprise the codebase are massive, containing several large components and duplicated or unused code that should be refactored to address serious problems and bugs that may arise further in development or production usage.

Suggestions

There are several structural errors that further complicate the already complex code, along with usage of deprecated or no longer recommended approaches to achieve certain functionality. It would be appropriate to refractor and adopt modern approaches and frameworks/tools that solve these issues. Further recommendations include:

- Refactoring the project structure to reduce complexity and improve clarity.
- Migrating the entire codebase to become fully strictly typed by describing all interfaces instead of using the “any” type (as well as ignoring compiler warnings and linting errors), which will resolve many serious internal errors and expected bugs in the codebase.

Refactoring

Code Structure

- The first and most important suggestion is to refactor the current project structure. If each specific user interface element or page has its own folder containing the components, styles, and helper code, it will be much easier to split the main components and files. Below is a possible example:

- Most of the files containing multiple very large components should be split into several smaller components, each within individual files, to avoid any code duplication and incorrectly duplicated functionality.
- Reusable components, functions, interfaces, etc. should be moved to shared folders.
- After restructuring, it is both important and critical to correctly use Storybook and create examples and definitions for isolation component testing and unit testing.

Typescript

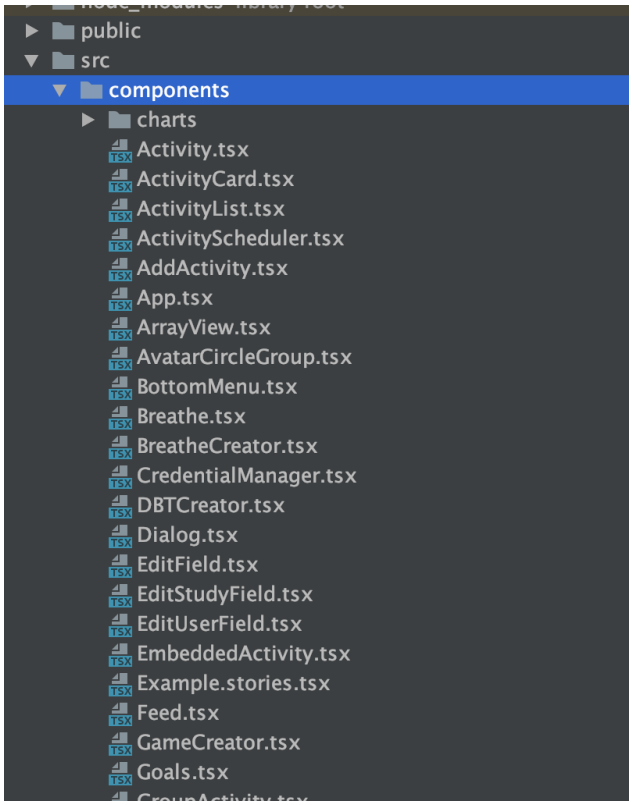
- All props that are received by function components should be completely described; an example is provided below:

```tsx
interface UserEvent {
  id: number;
  title: string;
  dateStart: string;
  dateEnd: string;
}

interface Props {
  event: UserEvent;
}

const EventItem: React.FC<Props> = ({ event }) => {
  const dispatch = useDispatch();
```

- Replace the “any” type one-by-one across each component, and if some props come, for example, from the “@material-ui/core” package, use the appropriate type definitions from this package.

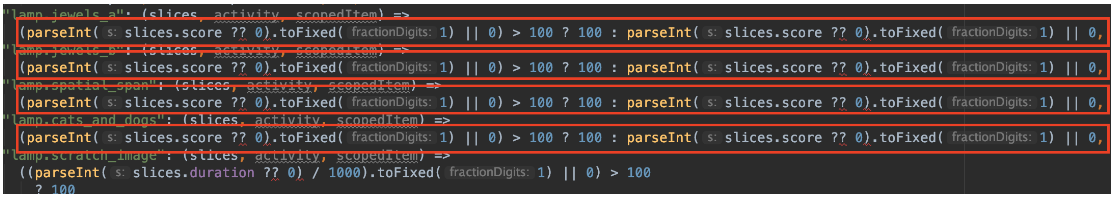
Project Structure

- The code structure has not ideally organized and does not have any kind of code-splitting, as all the logic is placed inside of components, and the whole project is a single flat folder.



- Due to the fact that components are overloaded with logic, styles, and functions, many of them are quite massive, which decreases code readability, developer productivity, and further dramatically increases the risk of bugs.

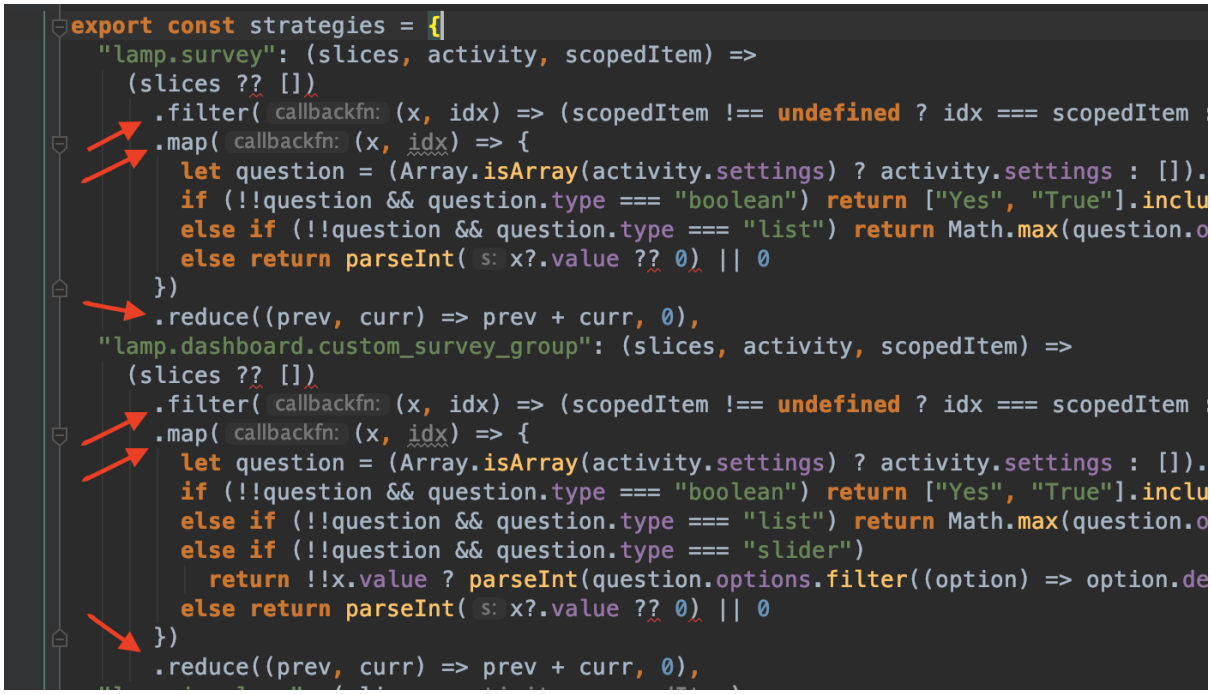
- Optimal size is no more than 200 lines, all files that are more than that should be refactored and split; large components with many state variables, effect functions, or rendered elements should also be split into smaller pieces as needed.
- Due to the fact that functions or variables are not placed in separate files for reusability, the “DRY principle" is not satisfied across this codebase. (The same logic is often duplicated even in a single component.)



In order to better organize the structure, it is necessary to transfer each component to a separate folder where its linked styles and components should also stored. If a single file is larger than ~250 lines, then it either needs to be split into multiple components or files, or both. All functions and variables that are reused in different places can be moved, for example, to a separate **common** folder, and re-used styles and interfaces should also be placed here as well.

React-specific Issues

- It is not the most correct approach to use array methods in a cascade one after another, as this makes the code difficult to understand and maintain; it is clearer to return the result to variables or refactor the code entirely using the “strategy pattern” (in the below example).



```
export const strategies = {
  "lamp.survey": (slices, activity, scopedItem) =>
    (slices ?? [])
      .filter( callbackfn: (x, idx) => (scopedItem !== undefined ? idx === scopedItem
      .map( callbackfn: (x, idx) => {
        let question = (Array.isArray(activity.settings) ? activity.settings : []).
        if (!!question && question.type === "boolean") return ["Yes", "True"].inclu
        else if (!!question && question.type === "list") return Math.max(question.o
        else return parseInt( s: x?.value ?? 0) || 0
      })
      .reduce((prev, curr) => prev + curr, 0),
  "lamp.dashboard.custom_survey_group": (slices, activity, scopedItem) =>
    (slices ?? [])
      .filter( callbackfn: (x, idx) => (scopedItem !== undefined ? idx === scopedItem
      .map( callbackfn: (x, idx) => {
        let question = (Array.isArray(activity.settings) ? activity.settings : []).
        if (!!question && question.type === "boolean") return ["Yes", "True"].inclu
        else if (!!question && question.type === "list") return Math.max(question.o
        else if (!!question && question.type === "slider")
          return !!x.value ? parseInt(question.options.filter((option) => option.de
        else return parseInt( s: x?.value ?? 0) || 0
      })
      .reduce((prev, curr) => prev + curr, 0),
```

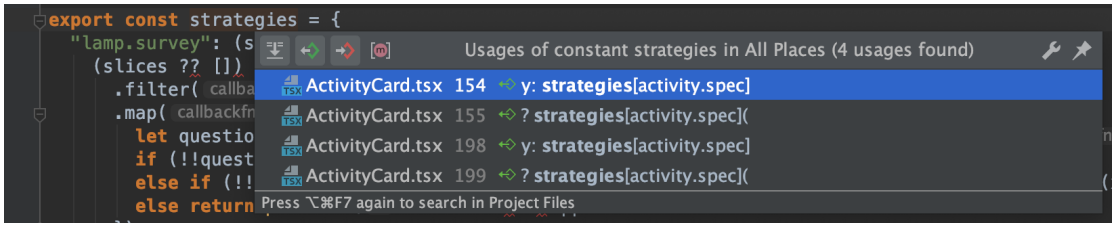
```
const earliestDate = () =>
  (activities || [])
    .filter( callbackfn: (x) => (selectedActivities || []).includes(x.name))
    .map( callbackfn: (x) => (activityEvents || {})[x.name] || [])
    .map( callbackfn: (x) => (x.length === 0 ? 0 : x.slice(0, 1)[0].timestamp))
    .sort( compareFn: (a, b) => a - b /* min */)
    .slice(0, 1)
    .map( callbackfn: (x) => (x === 0 ? undefined : new Date(x)))[0]
```

- Almost all components have variables that are declared but not used anywhere, and the list of ignored warnings is quite long; as a rule of thumb there should not be any warnings

or errors when testing OR building the codebase.

```
./src/components/NewMedication.tsx      webpackHotDevClient.js:138
  Line 10:3:    'Button' is defined but never used
@typescript-eslint/no-unused-vars
  Line 12:3:    'Container' is defined but never used
@typescript-eslint/no-unused-vars
  Line 17:3:    'DialogTitle' is defined but never used
@typescript-eslint/no-unused-vars
  Line 19:3:    'DialogActions' is defined but never used
@typescript-eslint/no-unused-vars
  Line 21:3:    'List' is defined but never used
@typescript-eslint/no-unused-vars
  Line 22:3:    'ListItem' is defined but never used
@typescript-eslint/no-unused-vars
  Line 23:3:    'Menu' is defined but never used
@typescript-eslint/no-unused-vars
  Line 24:3:    'ListItemText' is defined but never used
@typescript-eslint/no-unused-vars
  Line 25:3:    'MenuItem' is defined but never used
@typescript-eslint/no-unused-vars
  Line 26:3:    'Card' is defined but never used
@typescript-eslint/no-unused-vars
  Line 34:10:   'KeyboardTimePicker' is defined but never used
@typescript-eslint/no-unused-vars
  Line 34:30:   'KeyboardDatePicker' is defined but never used
@typescript-eslint/no-unused-vars
  Line 56:10:   'journalValue' is assigned a value but never used
@typescript-eslint/no-unused-vars
  Line 56:24:   'setJounalValue' is assigned a value but never used
@typescript-eslint/no-unused-vars
  Line 57:10:   'status' is assigned a value but never used
@typescript-eslint/no-unused-vars
  Line 57:18:   'setStatus' is assigned a value but never used
@typescript-eslint/no-unused-vars
  Line 58:10:   'anchorEl' is assigned a value but never used
@typescript-eslint/no-unused-vars
  Line 60:16:   'changeDate' is assigned a value but never used
@typescript-eslint/no-unused-vars
  Line 90:9:    'handleClose' is assigned a value but never used
@typescript-eslint/no-unused-vars
  Line 93:9:    'handleMenuItemClick' is assigned a value but never
used
@typescript-eslint/no-unused-vars
  Line 96:9:    'handleClick' is assigned a value but never used
```

- If a variable or a function is used only inside this component, it does not need to be exported; similarly if a variable or function is used in multiple places, it should be exported as part of a separate common file.



- If-else logic and conditions should not be using “==“ and instead should use “===“ and related operators for strict equality, as this is a significant source of data validation and conditional logic bugs. (This further applies to the use of “||” instead of “??” which is the correct operator for ternary conditional assignment.)

```
if (text.length != medicationName || "".length) {
```

```
if (medicationName != null && medicationName != "") {
  if (duration != null && duration != 0) {
```

- It is bad practice to declare global variables that do not need to be global, and these should instead use “let" (if being modified) and “const” (if immutable) instead of “var” (deprecated/outdated Javascript).

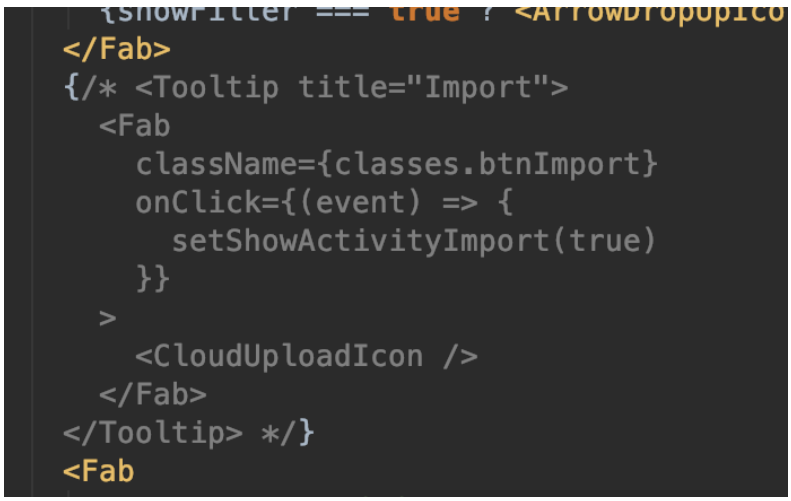
```
var weekday = ["Sunday", "Monday", "Tuesday", "Wednesd
var monthname = ["Jan", "Feb", "Mar", "Apr", "May", "J
```

- There are many cases of confusing and unclear code that uses several ternary operators that should instead be replaced by several statements or a function for readability.

```
data={events.map((d) => ({
  x: new Date(d.timestamp),
  y: strategies[activity.spec]
    ? strategies[activity.spec](
        activity.spec === "lamp.survey"
          ? d.temporal_slices
          : activity.spec === "lamp.scratch_image"
          ? d
          : d.static_data,
        selectedActivity,
        idx
      )
    : 0,
  slice: d.temporal_slices,
  missing:
    activity.spec === "lamp.survey"
      ? [null, "NULL"].includes( searchElement: d.temporal_slices[idx]?.value ?? null)
      : false, // sometimes the slice itself is missing, not set to null
```

- It is not ideal coding practice to leave commented code, console logs, and testing code in the codebase.

```
} else {
  if (selectedActivity.parentID !== x.studyID) {
    // let tag = await LAMP.Type.setAttachment(x.id, "me", "lamp.dashboard.activity_details", null)
    // console.dir("deleted tag " + JSON.stringify(tag))
    // await LAMP.Activity.delete(x.id)
    // result = (await LAMP.Activity.create(x.studyID, x)) as any
    // await LAMP.Type.setAttachment(result.data, "me", "lamp.dashboard.activity_details", {
    //   description: x?.description ?? "",
    //   photo: x?.photo ?? "",
    // })
  } else {
```



```
    {showFilter === true ? <ArrowDropUpIco
  </Fab>
  {/* <Tooltip title="Import">
    <Fab
      className={classes.btnImport}
      onClick={(event) => {
        setShowActivityImport(true)
      }}
    >
      <CloudUploadIcon />
    </Fab>
  </Tooltip> */}
  <Fab
```

```
const [anchor, handleOpen, handleClose] = useMenu()
/*const onAdd = React.useCallback(
  account => {
    OnAdd && OnAdd(account);
    handleClose();
  },
  [OnAdd, handleClose]
);

const onDelete = React.useCallback(
  id => {
    onRemove && onRemove(id);
    handleClose();
  },
  [onRemove, handleClose]
);

var [accounts, addAccount, deleteAccount] = useAccounts({
  accounts: Accounts, // Set initial accounts
  onDelete, // Callback fired on account delete
  onAdd // Callback fired on account add
});

const handleChangePassword = () => {
  onResetPassword && onResetPassword(anchor.id);
  handleClose();
};

const handleDelete = React.useCallback(() => deleteAccount(anchor.id), [
  anchor.id,
  deleteAccount
]);

const id = Math.max(...[-1, ...accounts.map(a => a.id)]) + 1;
const handleAdd = React.useCallback(
  (index = undefined) => addAccount(createTestAccount(id), index),
  [addAccount, id]
);

const handleClick = React.useCallback(
  ({ id, onClick }) => event => {
    id > 0 &&
      handleOpen(id, event);
    onClick &&
      onClick({ id, accounts, addAccount, deleteAccount, handleAdd });
  },
  [handleOpen, addAccount, deleteAccount, handleAdd, accounts]
);*/
let accounts = Accounts
```

```
export default function AddActivity({ ...p
  console.log(props.studyId)
  return (
```

- For the z-index property, it is advisable to try to use the smallest possible and declare them as constants in a shared file (if modifying z-index is required at all), otherwise, over time it will not be easily possible to track which elements can overlap others.

```
zIndex: 1111,  zIndex: 1000,
```

```
 zIndex: 111111,  zIndex: 99999,
```

```
zIndex: 2147483647,
```

- If a React element has an empty body, it should be a self-closed tag (i.e. “<Grid />” instead of “<Grid></Grid>”).

```
<Grid item xs={3} className={classes.lineyellow}></Grid>
<Grid item xs={3} className={classes.linegreen}></Grid>
<Grid item xs={3} className={classes.linered}></Grid>
<Grid item xs={3} className={classes.lineblue}></Grid>
```

- In the following cases, there is no need for the ternary operator, because the === operator returns true or false; similarly there are many cases across the codebase where

basic if-else logic and strict assertions are not ideally defined or not defined at all.

```
studyId === "" ? true : false{!!value || !!study ? true : false}
```

```
typeof text === "undefined" || (typeof text !== "undefined" && text?.trim() === "")
  ? true
  : false
```

- The value “undefined” should not be used as it is bad practice and leads to data faults, and instead, “null” should be used, although sparingly.

```
anchorEl: undefined,
credential: credential ?? undefined,
mode: mode ?? undefined,
```

- Many (if not all) linting errors are ignored instead of being resolved appropriately.

```
export default function Jewels({ onComplete, ...props }) {
  // eslint-disable-next-line
  const [jewels, setJewels] = useState(makeJewels())
  const [actions, setActions] = useState( initialState: [])
  // eslint-disable-next-line
  const [activity, setActivity] = useState( initialState: {})
```

- The “ “ element should not be used as a divider.

```
<TextField fullWidth={true} autoFocus margin="normal"
<div></div>
<TextField autoFocus margin="normal" variant="outlined
<Typography>{t( key: "Set reminder?")} </Typography>
{/* <Switch color="primary"> fsdfg</Switch> */}
<div></div>
<DatePicker label={t( key: "Start Date")} value={select
<div></div>
<DatePicker
```

- There are several places across the codebase where many redundant conditions are improperly declared, and they should instead be using “switch-case” syntax or a lookup dictionary.

```
if (props.goalType == "Exercise" || props.goalType == "Meditation" || props.
  setUnits( value: ["hours", "minutes"])
  setGoalUnit( value: "minutes")
} else if (props.goalType == "Weight") {
  setUnits( value: ["g", "Kg"])
  setGoalUnit( value: "Kg")
} else if (props.goalType == "Nutrition") {
  setUnits( value: ["mg", "g", "Ounces", "Pound", "Kg"])
  setGoalUnit( value: "Ounces")
} else if (props.goalType == "Sleep" || props.goalType == "Reading") {
  setUnits( value: ["hours", "minutes"])
  setGoalUnit( value: "hours")
} else if (props.goalType == "Finances") {
  setUnits( value: ["$", "€"])
  setGoalUnit( value: "$")
} else if (props.goalType == "Medication") {
  setUnits( value: ["mg", "g", "Ounces"])
  setGoalUnit( value: "mg")
} else {
  setUnits( value: ["Ounces", "mg", "g", "Pound", "Kg", "hours", "minutes", "
  setGoalUnit( value: "Ounces")
}
```

```
var goalIcon =
  props.goalType == "Exercise" ? (
    <Exercise />
  ) : props.goalType == "Weight" ? (
    <Weight />
  ) : props.goalType == "Nutrition" ? (
    <Nutrition />
  ) : props.goalType == "Meditation" ? (
    <BreatheIcon />
  ) : props.goalType == "Sleep" ? (
    <Sleeping />
  ) : props.goalType == "Reading" ? (
    <Reading />
  ) : props.goalType == "Finances" ? (
    <Savings />
  ) : props.goalType == "Mood" ? (
    <Emotions />
  ) : props.goalType == "Medication" ? (
    <Medication />
  ) : (
    <Custom />
  )
```

- Many variables and functions across the codebase are improperly named (i.e. “x” or “p” instead of “participant”), which could lead to significant bugs and coding errors and

decrease developer productivity as well.

```
let original = (await LAMP.ActivityEvent.allByParticipant(participant.id))
  .map( callbackfn: (x) => ({
    ...x,
    activity: _activities.find( predicate: (y) => x.activity === y.id),
  }))
  .filter( callbackfn: (x) => (!!x.activity ? !_hidden.includes( searchElement: `${x.timestamp}/${x.activity.id}`) : true))
  .sort( compareFn: (x, y) => x.timestamp - y.timestamp)
  .map( callbackfn: (x) => ({
    ...x,
    activity: (x.activity || { name: "" }).name,
  }))
  .groupBy( key: "activity") as any
let customEvents = _activities
  .filter( callbackfn: (x) => x.spec === "lamp.dashboard.custom_survey_group")
  .map( callbackfn: (x) =>
    x?.settings?.map((y, idx) =>
      original?.[y.activity]
        ?.map((z) => ({
          idx: idx,
          timestamp: z.timestamp,
          duration: z.duration,
          activity: x.name,
          slices: z.temporal_slices.find( predicate: (a) => a.item === y.question),
        }))
        .filter((y) => y.slices !== undefined)
    )
  )
  .filter( callbackfn: (x) => x !== undefined)
```

Typescript

The strict typing of data structure and interfaces in the codebase is practically absent, only sometimes present for primitive data types; there are only six total declared interfaces for the whole codebase, and in many other cases, either there is no typing or significant (incorrect) usage of the “any” / “as any” type.

```
export default function Activity({
  allActivities,
  activity,
  onSave,
  onCancel,
  details,
  studies,
  ...props
}: {
  allActivities?: any
  activity?: any
  onSave?: any
  onCancel?: any
  details?: any
  studies?: any
}) {
```

```
  ...props
}: {
  activities?: any
  value?: any
  onSave?: any
  onCancel?: any
  activitySpecId?: string
  details?: any
  studies?: any
  study?: any
}) {
  console.log(study)
```

```
export const CredentialManager: React.FunctionComponent<{
  id?: any
  onComplete?: any
  style?: any
  credential?: any
  mode?: string
}> = ({ id, onComplete, credential, mode, ...props }) => {
  const theme = useTheme()
  const [selected, setSelected] = useState<any>( initialState: {
```