subOS Redesign Proposal

A document to address the issues of the current design of subOS and to encourage the development of a new system

James Livesey (james@subnodal.com)

Working Draft: Tuesday, 15 December 2020
Review Draft: Sunday, 20 December 2020
Proposed Recommendation: Sunday, 20 December 2020
Complete Edition: Saturday, 13 February 2021

Contents

Introduction

Since the initial development of subOS in 2018, the design and technological issues surrounding the project have gradually grown over the years. Many of the issues were — at the time — out of our control; mainly because of the underlying technology surrounding the development of web standards. Having learnt from the underlying problems of the current design of subOS, it would be wise for a new, overhauled version of subOS to be made. We hope that in doing so, new opportunities will arise for us to redesign much of the operating system, from both a technological perspective and an interface design perspective, whilst maintaining our core values of what makes subOS unique.

About Subnodal

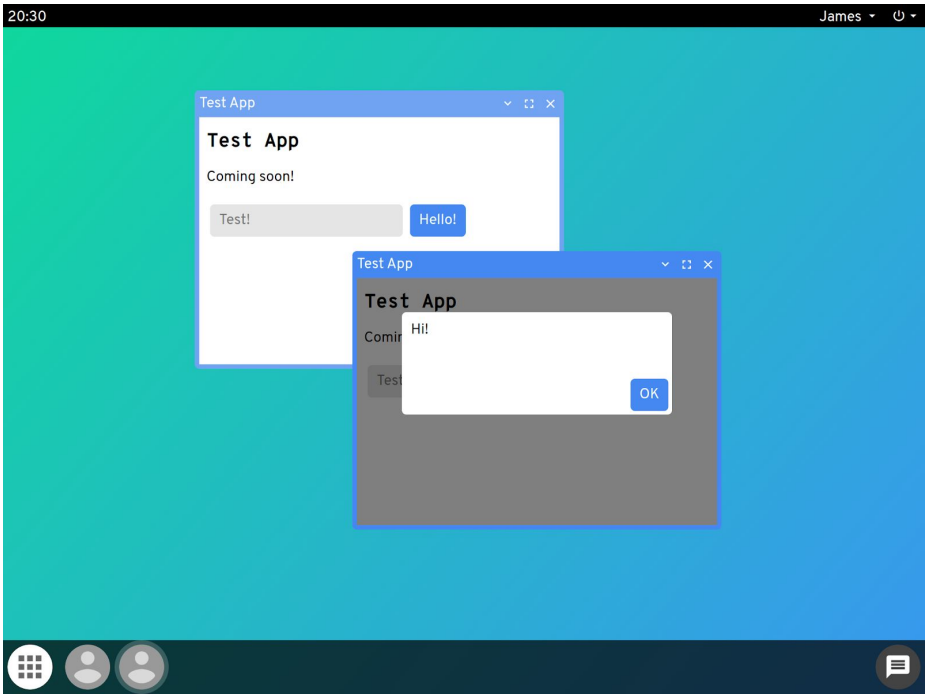

Figure 1: The desktop of the first subOS implementation.

Founded in 2018, Subnodal Technologies has since created multiple successful projects, most notably of which is the GameProxy website (gameproxy.host). In late 2018, the subOS project was started with the ambition to create an operating system built upon web technologies, such as HTML, CSS and JavaScript. The old implementation (**Figure 1**) was easily accessible via any web browser that is connected to the internet, and required no downloading of software to run the frontend. subOS additionally ran on real hardware, such as the Raspberry Pi. However, due to the age and increasing complexity of the project, subOS was abandoned for many months until Subnodal resurfaced in late 2019 and early 2020.

Problem diagnosis

What went wrong

There were many flaws in the old subOS which we should take steps to learn from:

- The operating system lacked relevant documentation about certain components (aside from information about screen switching, internationalisation etc. in the readme) for others to learn about the internals of the code
- The code itself wasn't sufficiently modular — the most ideal situation would be for components to be hot-swappable at runtime so that modules can then be edited without the need for restarting the system
- The code was not designed with object orientation in mind, relying too much on HTML, CSS and the Document Object Model system as opposed to JavaScript
- Applications made for subOS had to be written in an obscure way, causing HTML to be nested inside of JavaScript which is a non-standard practice and the separation between system code and application code was too blurred
- subOS was too dependent on third-party libraries such as jQuery, which under normal circumstances (such as for building a website) would be acceptable, but in the case of building an operating system, would not be suitable (it would be more beneficial for us to build our own libraries that perform similar functionality as their third-party counterparts to ensure greater compatibility)
- Much of the operating system was too reliant on its core developers, with subOS not reaching out to open-source developers enough
- Using Chromium as the runtime for subOS was bad for user experience since many keyboard controls would be taken by Chromium. A better approach would be to use a more customised developer-centric tool such as Electron, which is built on Chromium but has a greater amount of freedom with regards to interoperability
- subOS had no way to display most third-party websites since cross-site embedding relied on iframing, a practice which is blocked by popular websites such as Google using the `X-FRAME-OPTIONS` HTTP header. A better approach would be to use the `<webview>` element (exclusive to libraries such as Electron) which is not subject to the restrictions of the `<iframe>` element and gives subOS greater control over third-party content
- The interaction between the underlying Linux system and the subOS frontend was too loose — it would be more beneficial for subOS to be developed fully in JavaScript to enjoy the benefits of a Node.js backend, since the old implementation relied on a Python backend and a web server that bridged the backend and frontend (opening a potential security hole)

How we can improve

subOS can and should be improved in multiple ways:

- Intense documentation about each and every component and code section should be written so that contributors can fully understand all parts of the project
- Modularity should be brought to subOS so that components can be developed on-the-fly and by different teams within Subnodal, and so that the subOS codebase can efficiently use modules in multiple places (and in some cases, within applications)
- All of subOS's code should be written with the object-oriented paradigm in mind, which is beneficial due to our intended use of modularity. Particularly, user interface code should be written largely in JavaScript instead of HTML and CSS
- Equally, application code should be written fully in JavaScript, using distinct APIs exposed by the subOS environment for interactivity with other applications and system functionality, making the separation between the system and its running applications clearer
- subOS should be implementing many of its own libraries from the ground up, reducing and possibly eliminating its reliance on third-party libraries which are often intended for websites, can be insecure, and are often obscure in nature
- subOS should open up so that it can appeal to a wide community of open-source developers who specialise in full-stack development on a platform which is heavily oriented around JavaScript, Node.js and web standards
- subOS should use Electron as the runtime environment instead of Chromium, allowing for greater freedom over the underlying system that subOS runs on top of (such as allowing for direct access to the filesystem instead of having to communicate with the backend to do so)
- The `<webview>` element should be used where possible to serve up third-party content and the iframe-based structure used for applications should be replaced with a more integrated structure that does not rely on embedding, instead using the Web Worker API to allow for multithreading and to ensure code protection
- The Node.js backend offered by Electron should be used instead of the legacy Python web server backend to ease communication between the JavaScript frontend and now-JavaScript backend

Specifications for a new subOS

subOS's developers will overhaul the codebase of subOS, rewriting all of the code to meet our modern expectations of what a quality codebase should look like. The following sections outline the specifics of the new design of subOS.

Our core values

As with the old implementation of subOS, the new subOS will continue to uphold our values:

- A modern, secure, robust and open software architecture
- A system which can easily be used by anyone, no matter their age, language, disabilities or proficiency with technology
- A tool which is viable for everyday use, with strong application support and variety

Code layering

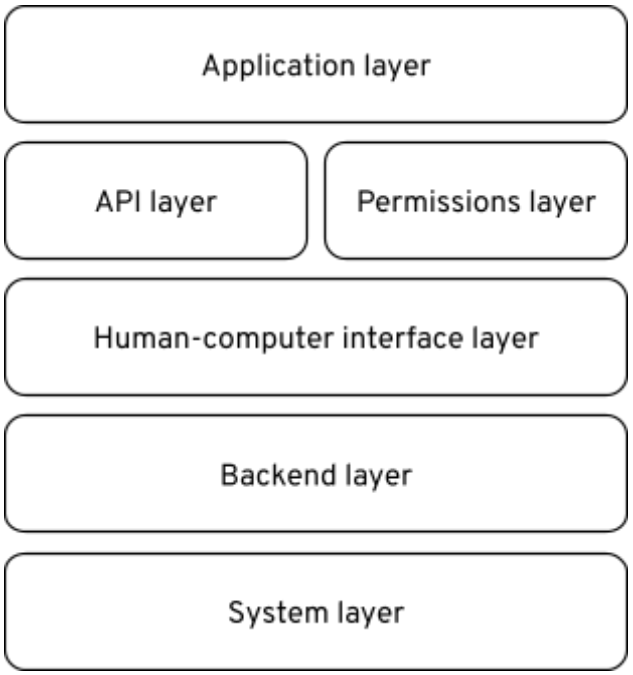

Figure 2: The various layers that will make up the subOS ecosystem.

The code of subOS will be split up into several distinct layers, as shown in **Figure 2**, to ensure that all code is easily maintainable. The layers are (from bottom to top):

- **System layer:** This is the underlying kernel that allows the subOS runtime to boot. On real hardware, this will be the Linux kernel — more specifically, using Debian or Raspbian (depending on the platform in use).
- **Backend layer:** This is the runtime for subOS that communicates with the system layer for hardware-specific operations such as storing/retrieving files, connecting to a network etc. The runtime then provides the human-computer interface layer and

then communicates with the API layer to run applications on the application layer. The runtime will be Electron, with the actual backend layer being implemented in JavaScript in the subOS code.
- **Human-computer interface layer:** This provides functionality for showing content on the user's screen, as well as for registering events with regards to reading user input such as from the keyboard/mouse.
- **API layer:** This exposes multiple APIs which allow applications in the application layer to communicate with the layers below, providing methods for using backend functionality.
- **Permissions layer:** This offers a permissions-based system to ensure that only certain applications (such as the desktop environment) can consume certain APIs (such as APIs for positioning UI controls on the human-computer interface layer, for example).
- **Application layer:** This is the least-trusted layer of subOS, and is where third-party applications are run. Third-party applications must use the permissions layer to use API functionality offered by the API layer. This is to ensure that untrusted applications can't make dangerous changes to the underlying system; otherwise an omission of such a feature would inevitably cause the creation of malware.

By following this layer-based system, we hope to see a more modular codebase where adding and removing dependencies across subOS's code is easier than before.

Modularity and use of the object orientation paradigm

The object-oriented programming paradigm will become increasingly beneficial as the development of subOS gradually changes over time. Using object orientation will also allow us to build better APIs which are easy to consume and integrate by applications that run on top of subOS.

To achieve full modularity with subOS, modules will be constructed in namespaces which will be provided by a library made by Subnodal. This namespace-based modularity system will provide a way for exposing only the functionality which is designed to be externally used by other modules, preventing code pollution.

For communication between modules, a management system will need to be devised so that modules can be loaded and unloaded accordingly. This will be designed in a similar style to npm — the Node.js package manager.

Software documentation

To ensure that all contributors to subOS fully understand the internals of the operating system, relevant and thorough documentation will be needed throughout so that it is easy for everyone to navigate the codebase. Subnodal will use a system which is based on code commenting, where each and every method in code will be given its own brief description, and — especially in the case of exposed parts of APIs — information about function

arguments and return values. Through the use of code commenting, we will then generate associated documentation files which will be visible and searchable online.

As well as documentation about the intricacies of specific code sections, documentation about the general structure of subOS will also be written, as well as various guides about how the users of subOS can become contributors. We will explore this later on in the proposal.

Separation of concerns

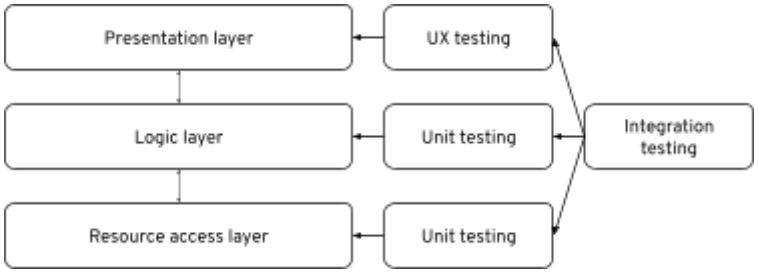

Figure 3: Concerns in applications and their surrounding testing procedures

In the context of subOS applications, and where applicable, a separation of concerns model will be used to make distinct the various aspects of an application's code. In the context of application development, a *concern* is any piece of information (such as the interface design of the application, and even trivial things such as the names of identifiers/variables) which plays an important role in the operation of a program.

Subnodal will incorporate the standard layout of concern separation into subOS, as shown in **Figure 3**:

- The **presentation layer** will house concerns surrounding the user interface of an application, such as the styling and theming of interface components to the positioning of controls such as buttons and text inputs.
- The **logic layer** will house concerns surrounding the application's implementation — the backend of the application, such as program states and operations carried out in the background.
- The **resource access layer** will house concerns surrounding the access (reading and writing) of data resources such as performing requests to web servers and REST APIs, loading imagery and other media for use within the program, as well as saving and opening documents created by the user.

The layers of concerns will be able to communicate with each other, such that the presentation layer can communicate with the logic layer, the logic layer can communicate with the resource access layer, and vice versa (as detailed in **Figure 3**). Communications between the presentation layer and the resource access layer should be avoided to prevent the distinct concerns from becoming too interconnected (and thus it would be hard for the separate layers to be replaced), and so the presentation layer and the resource access layer should use the logic layer as a path of communication.

For each of the layers of concerns, tests will be carried out to ensure that each layer meets quality standards (again, shown in **Figure 3**):

- The **presentation layer** will undergo **user experience (UX) testing** so that the user interface is easy-to-use by the end users of the application. UX testing should be considered as a superset of unit testing since all aspects of the user interface must be tested.
- The **logic layer** will undergo **unit testing** so that each function within the program logic works as intended and does not produce unexpected errors or side effects in operation.
- Likewise, the **resource access layer** will also undergo **unit testing** so that resources can be accessed by the program and so that documents made by the user are properly saved and written to.

Finally, to ensure that the application works as a whole, integration testing will be carried out to ensure that the communication between the layers of concerns works as intended. This should only be done once the individual unit tests have passed, so that we can be confident that the whole integration test works with the possibility of no errors. We will explore the implementation of unit and integration testing in the next specifications section.

How layers of concerns will be implemented

The separate layers of concerns will be implemented in separate files — and in some circumstances (such as if an application is sufficiently large or complex), with multiple files under any one layer. In most circumstances, the presentation layer will usually be implemented within a file named `presentation.js`, the logic layer will usually be implemented within `logic.js`, and the resource access layer will usually be implemented within `resources.js`.

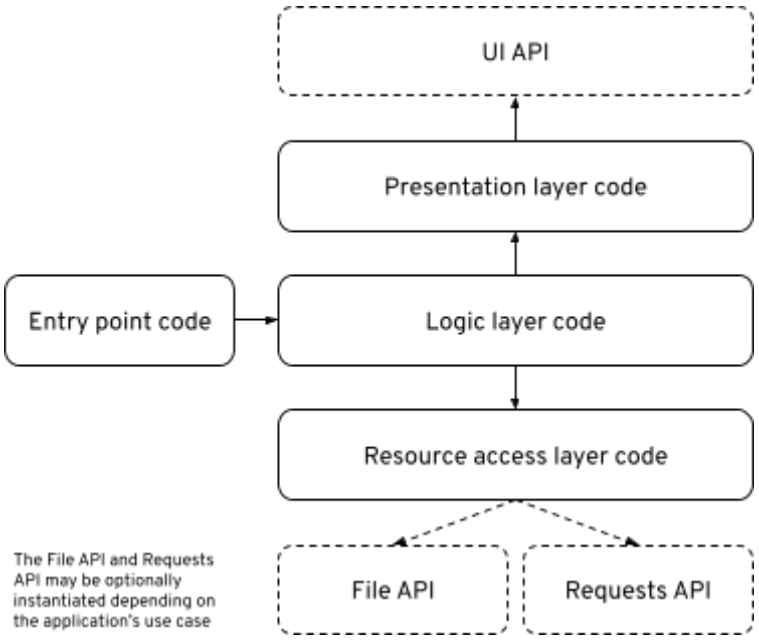

Figure 4: Layer dependency flow, with additional API dependencies listed

As an entry point, the code which initialises the logic layer will usually reside within `main.js` and will be run at application startup. Subsequently, the presentation and resource access layers will be initialised within the logic layer code, as shown in **Figure 4**. APIs such as subOS's UI API, File API and Requests API will be instantiated by their appropriate layers: the UI API will be instantiated by the presentation layer code, and the File API and Requests API will both optionally be instantiated by the resource access layer code.

Is the resource access layer needed?

In most cases, the resource access layer will be needed, even for activities which would otherwise usually seem trivial — such as the persistence of user preference and logic state. For very simple applications which don't need persistence, it is recommended that the application is configurable by the user in some way, so long as the configuration options are sensible and needed (for example, the ability to configure the typeface of text in a simple clock utility application may not be considered sensible, whereas configurability of the date/time format would). Otherwise — and only in exceptional circumstances — the resource access layer may be omitted.

Unit and integration testing

Within the scope of subOS, both unit and integration testing is key to quality assurance. To make this possible, a workflow will be adopted so that codebase changes can be tested during and after the development of a specific feature or issue fix.

Unit test files for JavaScript code will adopt the file extension of `.test.js` where the filename will match the filename of the code that is being tested. Similarly, integration tests will usually be named `main.test.js`.

Within the wider scheme of testing, quality assurance will be a quantitative measurement across multiple industry-standard properties:

- **Conceptual integrity:** How consistent the design is with respect to other components of the wider system.
- **Maintainability:** How easy it is to identify and implement fixes to issues.
- **Reusability:** How capable a particular module is with regards to its intent for reuse.
- **Integrality:** How capable a particular module is when used by or in conjunction with other modules.
- **Reliability:** How resistant a particular function is when handling errors and unexpected input.
- **Performance:** How responsive a particular function is given a time constraint.
- **Scalability:** How performant a particular function is when put under a large amount of stress with regards to input size and multithreaded load.
- **Security:** How capable a particular function is in preventing malicious exploitation.
- **Supportability:** How easy it is to document a particular function or module of the system.

- **Testability:** How easy it is to write and perform unit and integration tests on a particular module.
- **Usability:** How easy it is to use a particular system function as an end user.
- **Correctness:** How accurate and precise the output of certain functionality is.
- **Portability:** How capable a certain module or function is when changes to the environment are made, or when run in a different environment.

User interface design

We believe that the old interface layout for subOS continues to be unique and intuitive, and so we do not expect many further improvements over the old implementation of subOS regarding the placement of UI components. We do, however, believe that the style of subOS could be improved to incorporate some of the new UI design trends that have been popularised in recent years.

Use of shadows

The old implementation of subOS has only a few shadows on UI controls: one of the most obvious controls to have shadows is dropdown menus. We hope to include more drop shadows in the new implementation of subOS to make it easier for people to differentiate between overlapping panels — for example, in the old implementation of subOS, two inactive overlapping windows both bear the same colour on the window border, making it hard to determine where the edges of a window are when resizing the window. The new implementation seeks to resolve this issue by applying drop shadows to windows, as well as the top and bottom bars and menus.

However, we want to ensure that there's a balance between shadowed and non-shadowed controls, and so for the time being, we will only consider applying drop shadows to the aforementioned controls. This is so that we can introduce skeuomorphic hints into the design, whilst maintaining the modern flat look seen in many user interfaces today.

Further development on the specification of UI design will come later on in the project roadmap.

Glassmorphism

One of the most popular design trends of late 2020 and early 2021 is the *glassmorphism* design trend. This style can be applied to both light- and dark-mode themes and adds a subtle textured effect which resembles materials such as frosted acrylic. The panel that uses the design usually exhibits three main properties: a sufficient opacity to introduce background colour to the panel; a background blur to give the panel a frosted-look, and finally a drop shadow to make the panel 'lift' off of its parent in the Z axis.

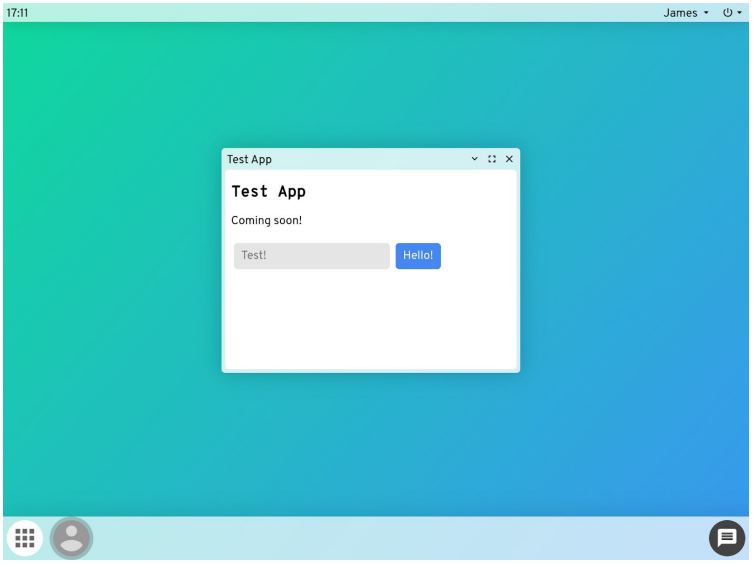

Figure 5a: Basic mockup of a light-mode glassmorphic effect made in the current implementation of subOS.

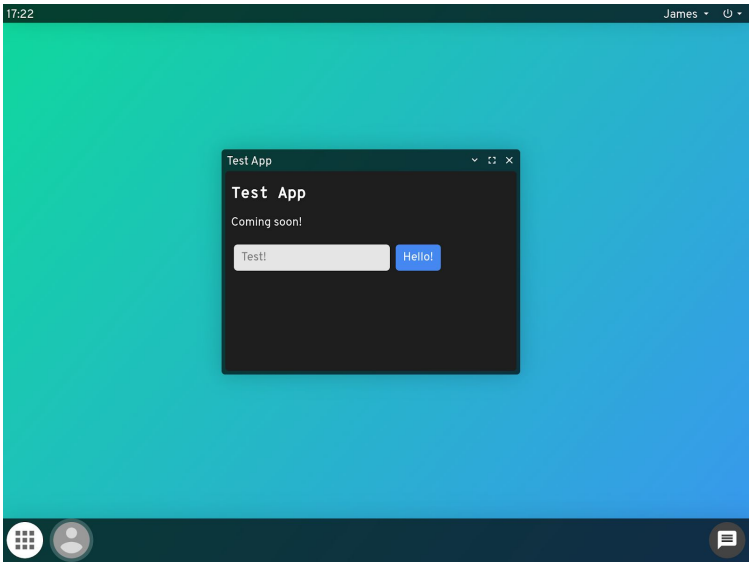

Figure 5b: Basic mockup of a dark-mode glassmorphic effect made in the current implementation of subOS.

Some somewhat low-fidelity examples of a light- and dark-mode glassmorphic effect design can be seen in **Figure 5a** and **Figure 5b**. Notice how the window title bars no longer adhere to the accent colour UI elements such as buttons — instead, the material of the info and app bars is used instead. This will allow us to use the window background as a surface to situate 'hangoff' panels such as a menu bar or tab layout which will integrate nicely with the window (not pictured).

We think that glassmorphism has a huge potential to enter into the design sphere due to how easy it is to incorporate the design. Subnodal hopes that subOS sets a leading example which many other products and applications will follow in the future.

Accessibility and glassmorphism

Despite glassmorphism being an eye-catching and appealing design trend, it does come with a few accessibility issues which can be resolved depending on how the glassmorphic effect is implemented. The accessibility issue in question is the lack of contrast between the panel background and its contents, and a high signal-to-noise ratio when a panel is overlaid across another panel containing content. This in turn can cause issues for vision-impaired users who may otherwise need a high-contrast theme to read text clearly.

To resolve this, panels incorporating glassmorphic design must:

- **Have an opacity of at least 60%:** This is to ensure that there is enough contrast between the panel's background and its content.
- **Have a blur of at least 10px:** This is to ensure that there is more discernible textual 'signal' from any 'noise' which is behind the panel.

All other accessibility guidelines surrounding contrast must be followed when considering glassmorphic design. Subnodal recommends that glassmorphic designs follow the Web Content Accessibility Guidelines (WCAG) 2.0 Level AA standards, and supplementary high-contrast themes follow the Level AAA standards.

Subnodal libraries external to subOS

For the sake of convenience, some libraries may be written in codebases external to subOS, with the respective libraries being 'imported into' the subOS runtime filesystem as a dependency via the subOS package manager. An example of a library which will reside in its own codebase is the internationalisation (i18n) library for subOS, which can additionally be used on the Subnodal website.

Openness to changes

As a key aspect to open-source operating system development, the ability to replace certain features of an operating system (such as the desktop environment) is regarded as a large benefit to free and open-source software (FOSS) by many. subOS hopes to achieve this openness through its extensive use of modules throughout the system, as well as the hotswappability of many applications (again, such as the desktop environment). This in turn will make the development of new features a more rapid process as community-suggested ideas can be easily integrated without the need for changes in multiple modules, as well as the need for frequently restarting the system to test new changes being largely eliminated.

Changes can later on be made within subOS using its own integrated development environment (IDE), which — for the purposes of subOS — will be a complimentary redesign of Codeslate, Subnodal's IDE. The modules affected by development changes will then be reloadable through the package manager.

Project roadmap

Development of the new subOS redesign project will happen in the following order, whether consecutively or concurrently:

1. Release of this document as a Proposed Recommendation
2. Updation of subnodal.com website to reflect the project's intentions
3. Decommissioning and archival of old subOS source code
4. Creation of code-commenting documentation system
5. Creation of module/namespace authoring system
6. Creation of unit and integration testing system
7. Creation of internationalisation (i18n) module
8. Finalisation and re-release of this document as a Complete Edition
9. Invitation to open-source community to welcome project contributors
10. Creation of new subOS codebase
    a. Creation of readme file, licence and other associated documentation
    b. Creation of base subOS system runtime
    c. Creation of subOS package manager (named subPack) alongside File API and Requests API
    d. Development of UI design specification
    e. Creation of application launcher and management module
    f. Creation of user management system and Users API
    g. Creation of permissions system
    h. Creation of command-line interface for launching applications
    i. Creation of UI API based on UI design specification
    j. Creation of subDE, the subOS Desktop Environment
    k. Development of desktop design mockups
    l. Creation of sign in screen
    m. Creation of base utility applications
11. Build of alpha release of subOS live image
12. Updation of website to reflect release of alpha image, in addition to release of subOS documentation for building applications
13. Continuation of development of subOS codebase for beta release
    a. Creation of subReader and other accessibility tools
    b. Creation of an application store, named the subStore
    c. Creation of more utility applications, such as a web browser
14. Further decision-making concerning new features to implement and continuation of roadmap

Ensuring that everybody can get involved

Developing a brand-new operating system — such as subOS — takes time and a huge number of contributions from the open-source community. To facilitate this, Subnodal will set up various schemes to bring new and learning contributors into the field of system design, computer science and operating system development. The details about these schemes are listed below.

Inclusion of a variety of contributors

Developing subOS is not just about the code that goes behind it — various contributors aside from software developers are needed to help the development of the operating system:

- **Systems architects:** As one of the most important roles in the undertaking of design in any system, systems architects are responsible for breaking down the development of subOS into smaller, more manageable subtasks which can then be handled by software engineers at an individual level. Their work is critical in ensuring that the communications between different teams of the subOS project is efficient.
- **User interface/graphic designers:** To create an environment which is appealing to potentially thousands of subOS users, both user interface designers and graphic designers are needed to create a modern, inviting and welcome look to the subOS user interface.
- **User experience designers:** To ensure that subOS is easy-to-use by everybody, user experience designers are key to the development of a friendly and familiar interface and general experience which hopes to make subOS manageable by people who — in the most impactful scenario — may have never even touched a computer before.
- **Accessibility experts:** So that subOS can be used by people who may need alternative ways to access and control their devices, accessibility experts will be needed to allow for potentially thousands more users so that they can enjoy subOS as equally as other, possibly more able users.
- **Linguists and translators:** Because Subnodal hopes to appeal to a wider, more global market, linguists and translators will be a big help in the expansion of subOS into international markets. Subnodal will be looking for many translators so that subOS can be operated in most — ideally all — languages spoken in the world today.
- **Technical evangelists:** To reach out to developers and potential contributors — whether new or experienced to operating system development — around the world, Subnodal welcomes technical evangelists who promote and share subOS with others. This will in turn let subOS grow as the community surrounding it widens.

With the inclusion of roles across multiple disciplines, it is necessary for Subnodal to disclose the fact that not everybody needs to be an able coder in order to collaborate with the subOS project — we ask that those who wish to help be familiar with many of the basic concepts of operating system development and — at the very least — digital literacy.

Allowing new developers to grow

So that subOS can truly be developed by a diverse and talented group of contributors, Subnodal hopes to introduce people who are new to software development and programming into the world of computer science and software engineering. Even though the development of a large operating system is a demanding task, Subnodal will allow people who are first-time contributors to join in through simple guides that will help them to grow to become frequent maintainers:

1. **Recognition of contributor expectations:** Subnodal will provide a Code of Conduct for all contributors to follow, allowing the subOS development community to remain peaceful and respectful.
2. **Instructions for replicating subOS:** Within the relevant documentation, Subnodal will ensure that potential contributors will be able to build their own instances of subOS and create their own environments for developing many aspects of the operating system.
3. **Reporting issues with the subOS code:** To make sure that subOS works as intended, Subnodal will encourage its contributors to open issues about bugs and other problems they find within the system. Subnodal will then let other, more experienced developers respond to issues or — if the original issue reporter feels up to contributing — let the contributor who made the issue attempt to fix the problems and bugs themselves.
4. **Documenting the functionality of subOS:** Contributors who are learning about the intricacies of the codebase that makes up subOS will be able to contribute by creating pull requests which document code modules through code comment annotation in addition to the creation of dedicated documentation guides. This is ideal for those who are becoming familiar with JavaScript through their experience of other programming languages and who are eager to learn about what the various modules of subOS do.
5. **Suggesting new features to add to subOS:** Once contributors have become familiar with the codebase, feature suggestion is recommended for contributors who wish to make a big positive impact to subOS. Through listening to feedback from the community, experienced contributors can start to assemble the ideas for creating new features which can then be implemented into subOS. This is particularly suited to systems architects who wish to tackle what would — in some cases — be originally a large feature.
6. **Implementing features into the subOS codebase:** Those who are more willing to continually develop and expand the subOS codebase are then welcome to help with the implementation of new feature ideas which have been suggested by the community. This requires contributors to have a large, widespread knowledge of the subOS codebase and to have the ability to write clean, easy-to-understand code.
7. **Managing teams within the subOS contributor community:** The most experienced of the subOS contributors will help to manage the various teams that collectively assemble the features of subOS and help new contributors to get into the development community.

Further reading

- *Object-oriented JavaScript for beginners*; several contributors on MDN
- *Software Architecture & Design Introduction*; Tutorialspoint
- *A Node.js Guide to Actually Doing Integration Tests*; Blažecki on Toptal
- *Web Content Accessibility Guidelines (WCAG) 2.0*; several contributors on W3C
- *Glassmorphism CSS Generator*; HYPE4
- *Glassmorphism in user interfaces*; Malewicz on UX Collective
- *Acrylic material*; several contributors on Microsoft Docs
- *Contrast and Color Accessibility*; WebAIM