Перевод статьи для Хабра с русского на английский:

В Бегете мы долго и успешно занимаемся виртуальным хостингом, используем много OpenSource-решений, и теперь настало время поделиться с сообществом нашей разработкой: файловым менеджером Sprut.IO, который мы разрабатывали для наших пользователей и который используется у нас в панели управления. Приглашаем всех желающих присоединиться к его разработке. О том, как он разрабатывался и почему нас не устроили существующие аналоги, какие костыли технологии мы использовали и кому он может пригодиться, расскажем в этой статье.

Сайт проекта:https://sprut.io

Демо доступно по ссылке: https://demo.sprut.io:9443

Исходный код: https://github.com/LTD-Beget/sprutio

Зачем изобретать свой файловый менеджер

В 2010 году мы использовали NetFTP, который вполне сносно решал задачи открыть/загрузить/подправить несколько файлов.

Однако, пользователям иногда хотелось научиться переносить сайты между хостингами или у нас между аккаунтами, но сайт был большой, а интернет у пользователей не самый хороший. В итоге, или мы делали это сами (что явно было быстрее), или объясняли, что такое SSH, MC, SCP и прочие страшные вещи.

Тогда у нас и появилась идея сделать WEB двух-панельный файловый менеджер, который работает на стороне сервера и может копировать между разными источниками на скорости сервера, а также, в котором будут: поиск по файлам и директориям, анализ занятого места (аналог ncdu), простая загрузка файлов, ну и много всего интересного. В общем, все то, что облегчило бы жизнь нашим пользователям и нам.

В мае 2013 мы выложили его в продакшн на нашем хостинге. В некоторых моментах получилось даже лучше, чем мы хотели изначально — для загрузки файлов и доступа к локальной файловой системе написали Java апплет, позволяющий выбрать файлы и все сразу скопировать на хостинг или наоборот с хостинга (куда копировать не так важно, он умел работать и с удаленным FTP и с домашней директорией пользователя, но, к сожалению, скоро браузеры не будут его поддерживать).

Прочитав на Хабре про аналог, мы решили выложить в OpenSource наш продукт, который получился, как нам кажется, отличным работающим и может принести пользу. На отделение его от нашей инфраструктуры и приведение к подобающему виду ушло еще девять месяцев. Перед новым 2016 годом мы выпустили Sprut.IO.

Как он работает

Делали для себя и использовали самые, по нашему мнению, новые, стильные, молодежные инструменты и технологии. Часто использовали то, что было уже для чего-то сделано.

Есть некоторая разница в реализации Sprut.IO и версии для нашего хостинга, обусловленная взаимодействием с нашей панелью. Для себя мы используем: полноценные очереди, MySQL, дополнительный сервер авторизации, который отвечает и за выбора конечного сервера, на котором располагается клиент, транспорт между нашими серверами по внутренней сети и так далее.

Sprut.IO состоит из нескольких логических компонентов:

1) web-морда,

2) nginx+tornado, принимающие все обращения из web,

3) конечные агенты, которые могут быть размещены как на одном, так и на многих серверах.

Фактически, добавив отдельный слой с авторизацией и выбором сервера, можно сделать мультисерверный файловый менеджер (как в нашей реализации). Все элементы логически можно поделить на две части: Frontend (ExtJS, nginx, tornado) и Backend (MessagePack Server, Sqlite, Redis).

Схема взаимодействия представлена ниже:

Frontend

Web интерфейс — все достаточно просто, ExtJS и много-много кода. Код писали на CoffeeScript. В первых версиях использовали LocalStorage для кеширования, но в итоге отказались, так как количество багов превышало пользу. Nginx используется для отдачи статики, JS кода и файлов через X-Accel-Redirect (подробно ниже). Остальное он просто проксирует в Tornado, который, в свою очередь, является своеобразным роутером, перенаправляя запросы в нужный Backend. Tornado хорошо масштабируется и, надеемся, мы выпилили все блокировки, которые сами же и наделали.

Backend

Backend состоит из нескольких демонов, которые, как водится, умеют принимать запросы из Frontend. Демоны располагаются на каждом конечном сервере и работают с локальной файловой

системой, загружают файлы по FTP, выполняют аутентификацию и авторизацию, работают с SQLite (настройки редактора, доступы к FTP серверам пользователя).

Запросы в Backend отправляются двух видов: синхронные, которые выполняются относительно быстро (например, листинг файлов, чтение файла), и запросы на выполнение каких-либо долгих задач (загрузка файла на удаленный сервер, удаление файлов/директорий и т.п.).

Синхронные запросы — обычный RPC. В качестве способа сериализации данных используется msgpack, который хорошо зарекомендовал себя в плане скорости сериализации/десериализации данных и поддержки среди других языков. Также рассматривали python-специфичный rfoo и гугловский protobuf, но первый не подошел из-за привязки к python (и к его версиям), а protobuf, с его генераторами кода, нам показался избыточным, т.к. число удаленных процедур не измеряется десятками и сотнями и необходимости в выносе API в отдельные proto-файлы не было.

Запросы на выполнение долгих операций мы решили реализовать максимально просто: между Frontend и Backend есть общий Redis, в котором хранится выполняемый таск, его статус и любые другие данные. Для запуска задачи используется обычный синхронный RPC-запрос. Flow получается такой:

Frontend кладет в редис задачу со статусом «wait»

Frontend делает синхронный запрос в backend, передавая туда id задачи

Backend принимает задачу, ставит статус «running», делает fork и выполняет задачу в дочернем процессе, сразу возвращая ответ на backend

Frontend просматривает статус задачи или отслеживает изменение каких-либо данных (например, количество скопированных файлов, которое периодически обновляется с Backend).

Интересные кейсы, которые стоит упомянуть

Загрузка файлов с Frontend

Задача:

Загрузить файл на конечный сервер, при этом Frontend не имеет доступа к файловой системе конечного сервера.

Решение:

Для передачи файлов msgpack-server не подходил, основная причина была в том, что пакет не мог быть передан побайтово, а только целиком (его надо сначала полностью загрузить в память и только потом уже сериализовывать и передавать, при большом размере файла будет OOM), в итоге решено было использовать отдельного демона для этого.

Процесс операции получился следующий:

Мы получаем файл от nginx, пишем его в сокет нашего демона с заголовком, где указано временное расположение файла. И после того, как файл полностью передан, отправляем запрос в RPC на перемещение файла в конечное расположение (уже к пользователю). Для работы с сокетом используем пакет pysendfile, сам сервер самописный на базе стандартной питоновской библиотеки asyncore

Определение кодировки

Задача:

Открыть файл на редактирование с определением кодировки, записать с учетом исходной кодировки.

Проблемы:

Если у пользователя некорректно распознавалась кодировка, то при внесении изменений в файл с последующей записью мы можем получить UnicodeDecodeError и изменения не будут записаны.

Все «костыли», которые в итоге были внесены, являются итогом работы по тикетам с файлами, полученными от пользователей, все «проблемные» файлы мы также используем для тестирования после внесенний изменений в код.

Решение:

Исследовав интернет в поисках данного решения, нашли библиотеку chardet. Данная библиотека, в свою очередь, является портом библиотеки uchardet от Mozilla. Она, например, используется в известном редакторе https://notepad-plus-plus.org

Протестировав ее на реальных примерах, мы поняли, что в реальности она может ошибаться. Вместо CP-1251 может выдаваться, например, «MacCyrillic» или «ISO-8859-7», а вместо UTF-8 может быть «ISO-8859-2» или частный случай «ascii».

Кроме этого, некоторые файлы на хостинге были utf-8, но содержали странные символы, то ли от редакторов, которые не умеют корректно работать с UTF, то ли еще откуда, специально для таких случаев также пришлось добавлять «костыли».

Пример распознавания кодировки и чтения файлов, с комментариями

Параллельный поиск текста в файлах с учетом кодировки файла

Задача:

Организовать поиск текста в файлах с возможностью использования в имени «shell-style wildcards», то есть, например, 'pupkin@*.com' '$* = 42;' и т.д.

Проблемы:

Пользователь вводит слово «Контакты» — поиск показывает, что нет файлов с данным текстом, а в реальности они есть, но на хостинге у нас встречается множество кодировок даже в рамках одного проекта. Поэтому поиск также должен учитывать это.

Несколько раз столкнулись с тем, что пользователи по ошибке могли вводить любые строки и выполнять несколько операций поиска на большом количестве папок, в дальнейшем это приводило к возрастанию нагрузки на серверах.

Решение:

Многозадачность организовали достаточно стандартно, используя модуль multiprocessing и две очереди (список всех файлов, список найденных файлов с искомыми вхождениями). Один воркер строит список файлов, а остальные, работая параллельно, разбирают его и осуществляют непосредственно поиск.

Искомую строку можно представить в виде регулярного выражения, используя пакет fnmatch. Ссылка на итоговую реализацию поиска.

Для решения проблемы с кодировками приведен пример кода с комментариями, там используется уже знакомый нам пакет chardet.

Пример реализации воркера

В итоговой реализации добавлена возможность выставить время выполнения в секундах (таймаут) — по умолчанию выбран 1 час. В самих процессах воркеров понижен приоритет выполнения для снижения нагрузки на диск и на процессор.

Распаковка и создание файловых архивов

Задача:

Дать пользователям возможность создавать архивы (доступны zip, tar.gz, bz2, tar) и распаковывать их (gz, tar.gz, tar, rar, zip, 7z)

Проблемы:

Мы встретили множество проблем с «реальными» архивами, это и имена файлов в кодировке cp866 (DOS), и обратные слеши в именах файлов (windows). Некоторые библиотеки (стандартная ZipFile python3, python-libarchive) не работали с русскими именами внутри архива. Некоторые реализации библиотек, в частности SevenZip, RarFile не умеют распаковывать пустые папки и файлы (в архивах с CMS они встречаются постоянно). Также пользователи всегда хотят видеть процесс выполнения операции, а как это сделать если не позволяет библиотека (например просто делается вызов extractall())?

Решение:

Библиотеки ZipFile, а также libarchive-python пришлось исправлять и подключать как отдельные пакеты к проекту. Для libarchive-python пришлось сделать форк библиотеки и адаптировать ее под python 3.

Создание файлов и папок с нулевым размером (баг замечен в библиотеках SevenZip и RarFile) пришлось делать отдельным циклом в самом начале по заголовкам файлов в архиве. По всем багам разработчикам отписали, как найдем время то отправим pull request им, судя по всему, исправлять они это сами не собираются.

Отдельно сделана обработка gzip сжатых файлов (для дампов sql и проч.), тут обошлось без костылей с помощью стандартной библиотеки.

Прогресс операции отслеживается с помощью вотчера на системный вызов IN_CREATE, используя библиотеку pyinotify. Работает, конечно, не очень точно (не всегда вотчер срабатывает, когда большая вложенность файлов, поэтому добавлен магический коэффициент 1.5), но задачу отобразить хоть что-то похожее для пользователей выполняет. Неплохое решение, учитывая, что нет возможности отследить это, не переписывая все библиотеки для архивов.

Код распаковки и создания архивов.

Пример кода с комментариями

Повышенные требования к безопасности

Задача:

Не дать пользователю возможности получить доступ к конечному серверу

Проблемы:

Все знают, что на хостинговом сервере одновременно могут находиться сотни сайтов и пользователей. В первых версиях нашего продукта воркеры могли выполнять некоторые операции с root-привилегиями, в некоторых случаях теоретически (наверное) можно было получить доступ к чужим файлам, папкам, прочитать лишнее или что-то сломать.

Конкретные примеры, к сожалению, привести не можем, баги были, но сервер в целом они не затрагивали, да и являлись больше нашими ошибками, нежели дырой в безопасности. В любом случае, в рамках инфраструктуры хостинга есть средства снижения нагрузки и мониторинга, а в версии для OpenSource мы решили серьезно улучшить безопасность.

Решение:

Все операции были вынесены, в так называемые, workers (createFile, extractArchive, findText) и т.д. Каждый worker, прежде чем начать работать, выполняет PAM аутентификацию, а также setuid пользователя.

При этом все воркеры работают каждый в отдельном процессе и различаются лишь обертками (ждем или не ждем ответа). Поэтому, даже если сам алгоритм выполнения той или иной операции может содержать уязвимость, будет изоляция на уровне прав системы.

Архитектура приложения также не позволяет получить прямой доступ к файловой системе, например, через web-сервер. Данное решение позволяет достаточно эффективно учитывать нагрузку и мониторить активность пользователей на сервере любыми сторонними средствами.

Установка

Мы пошли по пути наименьшего сопротивления и вместо ручной установки подготовили образы Docker. Установка по сути выполняется несколькими командами:

```
user@host:~$ wget https://raw.githubusercontent.com/LTD-Beget/sprutio/master/run.sh
user@host:~$ chmod +x run.sh
user@host:~$ ./run.sh
```

run.sh проверит наличие образов, в случае если их нет скачает, и запустит 5 контейнеров с компонентами системы. Для обновления образов необходимо выполнить

```
user@host:~$ ./run.sh pull
```

Остановка и удаление образов соответственно выполняются через параметры stop и rm. Dockerfile сборки есть в коде проекта, сборка занимает 10-20 минут.

Как поднять окружение для разработки в ближайшее время напишем на сайте и в wiki на github.

Помогите нам сделать Sprut.IO лучше

Очевидных возможностей для дальнейшего улучшения файлового менеджера достаточно много.

Как наиболее полезные для пользователей, нам видятся:

Добавить поддержку SSH/SFTP

Добавить поддержку WebDav

Добавить терминал

Добавить возможность работы с Git

Добавить возможность расшаривания файлов

Добавить переключение тем оформление и создание различных тем

Сделать универсальный интерфейс работы с модулями

Если у вас есть дополнения, что может быть полезно пользователям, расскажите нам о них в комментариях или в списке рассылки sprutio-ru@groups.google.com.

Мы начнем их реализовывать, но не побоюсь этого сказать: своими силами на это уйдут годы если не десятилетия. Поэтому, если вы хотите научиться умеете программировать, знаете Python и ExtJS и хотите получить опыт разработки в открытом проекте — приглашаем вас присоединиться к разработке Sprut.IO. Тем более, что за каждую реализованную фичу мы будем выплачивать вознаграждение, так как нам не придется реализовывать ее самим.

Список TODO и статус выполнения задач можно увидеть на сайте проекта в разделе TODO.

Спасибо за внимание! Если будет интересно, то с радостью напишем еще больше деталий про организацию проекта и ответим на ваши вопросы в комментариях.

Сайт проекта: https://sprut.io

Демо доступно по ссылке: https://demo.sprut.io:9443

Исходный код: https://github.com/LTD-Beget/sprutio

Русский список рассылки: sprutio-ru@groups.google.com

Английский список рассылки: sprutio@groups.google.com

Перевод:

Development → Web file manager Sprut.IO в OpenSource

Here at BeGet we have a long and successful history of working with virtual web hosting and implementing various open-source solutions and now it's time to share our invention with the world: file manager Sprut.IO, which we developed for our clients and which is used in our control panel. Please feel invited to join its development. In this article we will tell you how it was developed, why we weren't pleased with existent file managers, which [kludges] technologies we used and who can profit from it.

Project website:https://sprut.io

Demo version available at: https://demo.sprut.io:9443

Source code: https://github.com/LTD-Beget/sprutio

[Картинка]

Why invent your own file manager

In 2010 we were using NetFTP, which handled such tasks as opening/loading/editing quite bearably.

However from time to time our users wanted to learn how to move websites between web hostings or between our accounts, but the website was too large and the internet was not the best. In the end we either had to do it ourselves (whish was obviously faster) or to explain what SSH, MC, SCP and other horrendous things are.

That's when we had the idea to create an orthodox WEB file manager, working on the server's site, which would be able to copy between different sources with server speed and would offer: file and

directory search, a disk usage analyzer (an analogue of ncdu), simple file uploading and a lot of other great stuff. Let's say everything that would make our users' and our lives easier.

In May 2013 the file manager went into production on our web hosting. Some parts even turned out to be better than we originally planned: we created a Java applet for file uploading and access to the local file system, which allowed to select files and copy them to the web hosting or from it all at once (the destination is irrelevant, the file manager could work both with remote FTP and user's home directory, unfortunately soon browsers won't support it anymore).

After reading about a comparable manager on Habrahabr, we decided to publish ours as an open-source product, which, as we think, proved itself to be [amazing] useful and highly functional. We spent another 9 months separating it from our infrastructure and whiping it into shape. Just before NYE 2016 we released Sprut.IO.

How it works

We created it for ourselves and believe to have implemented the latest, most stylish and youthful tools and technologies. Often we used tools that had already been developed previously.

There's a certain difference between the realization of Sprut.IO and the version we created for our web hosting, meant to interact with our panel. For ourselves we use: full-featured queues, MySQL, an additional authorization server, which is also responsible for selection of the endpoint server, where the client is, transport between our servers and the internal network and so on.

Sprut.IO contains several logical components:

1) web-face,

2) nginx+tornado, receiving all web invocations,

3) endpoint agents, that can be located both on one or on several servers.

Basically you can create a multiserver file manager by adding a separate layer with authorization and server choice (as we did in our realization). All elements can be logically separated into two parts: Frontend (ExtJS, nginx, tornado) and Backend (MessagePack Server, Sqlite, Redis).

See the interaction scheme below:

[Картинка]

Frontend

Web interface; the whole thing is quite simple, ExtJS and a looot of code. Code written on CoffeeScript. In the first versions we used LocalStorage for caching, but in the end we decided against it since there were more bugs than benefit. Nginx is used to serve static content, JS code and files via X-Accel-Redirect (read more below). The rest is just proxied to Tornado, which in turn is a kind of router, redirecting queries to the correct Backend. Tornado can be perfectly scaled and we hope to have killed off all blockings we had created ourselves.

Backend

Backend consists of several demons that, as usual, can receive queries from Frontend. The demons are located on each endpoint server and work with the local file system, upload files via FTP, perform authentication and authorization, work with SQLite (editor settings, access to user's FTP servers).

There are two types of queries sent to Backend: synchronous, executed relatively fast (e.g. file listing or reading) and queries for execution of long tasks (uploading file to remote server, deleting files/directories etc.).

Synchronous queries are a usual RPC. Data serialization is realized via msgpack, which showed itself to be excellent in data serialization/deserialization speed and language support. We did also consider rfoo for Python and Google's protobuf, but while the first one didn't suit because of its tie to Python (and its versions), we thought the second one was excessive since there aren't dozens and hundreds of remote procedures and there was no need intaking the API out to separate proto-files.

We decided to realize queries for long operations as easily as possible: There's a shared Redis between Frontend and Backend, which contains the executed task, its status and any other information. The task is started with a common synchronous RPC query. The flow looks like this:

1. Frontend puts a task with the status "wait" into Redis

2. Frontend sends a synchronous query to Backend, handing over the task ID

3. Backend receives the task, sets the status "running", uses fork and executes the task in a child process, sending a response right to Backend

4. Frontend views the task's status or controls changes of any data (such as number of copied files, which is being updated by Backend from time to time).

Interesting cases, worth mentioning

Uploading files from Frontend

Task:

Upload files to endpoint server while Frontend does not have access to the endpoint server's file system.

Solution:

Msgpack server didn't suit for file transfer, mainly because the package could not be transferred by bytes, but only at once (first it needs to be completely uploades to the memory and only then be serialized and transferred, in case of a large file you'd be OOM), so in the end we decided to use a separate daemon for that.

The operation process looks as following:

We receive a file from Nginx, write it into our daemon socket with a title, containing the temporary file location. After completion of the file transfer we send a query to RPC to transfer the file to its final location (to the user). To work with the socket we use the pysendfile package, the server is selfwritten, based on the standard Python library asyncore.

Charset recognition

Task:

Open file for editing, recognizing the charset, writing it, considering the original charset.

Problems:

If the charset hasn't been correctly recognized on the user's computer, then editing the file, we could receive UnicodeDecodeError next time, so that changes wouldn't be saved.

All final "kludges" are result of work with file tickets, received from users, all "problematic" files are being used for testing after editing the code.

Solution:

Having combed through the internet, looking for a solution, we found the chardet library. This library, in turn, is a port of the uchardet library by Mozilla. It's being used in such famous projects as the editor https://notepad-plus-plus.org

Having tested it under various operating conditions, we noticed that in reality it could err. For instance it might show "MacCyrillic" or "ISO-8859-7" instead of CP-1251 and "ISO-8859-2" or often "ASCII" instead of UTF-8.

Again, some UTF-8 files on the web hosting contained weird characters, either from editors that didn't process UTF correctly or from somewhere else. However these cases also required new "kludges".

Example of charset recognition and file reading, with comments

Parallel charset-aware text search in files

Task:

Organize text search in files with the option of using "shell-style wildcards" in the name, that is for instance 'pupkun@*com' '$* = 42;' etc.

Problems:

The user searches for "Contacts", but the search result shows no files with this text while actually they do exist. It's just that there can be many different charsets on the web hosting, even within one project, so that the search needs to consider this as well.

A couple of times we observed a situation when by mistake users were able to put in any lines and perform some search operations in a big amount of folders, which led to a load increment on servers.

Solution:

We organized multitasking quite commonly by using the multiprocessing module and two queues (list of all files, list of found files with included listings). One worker builds the file list, the others review it at the same time and perform the search itself.

The searched line can be imagined as a regular expression, using the fnmatch package. Link to final implementation.

To solve the charset issue see a code example with comments, where we used the chardet package, mentioned earlier.

> Worker implemention example

The final implementation contains the additional option of setting execution time in seconds (timeout), set to 1 hour by default. The execution priority is lowered in the worker processes to drop the disk and processor load.

Unpacking and creating file archives

Task:

Give users a possibility to create archives (zip, tar.gz, bz2, tar available) and to unpack them (gz, tar.gz, tar, rar, zip, 7z).

Problems:

We faced a multitude of problems with "real" archives, such as file names in cp866 charset (DOS) and backslashes in filenames (windows). Some libraries (standard ZipFile python3, python-libarchive) didn't work with cyrillic names within the archive. Some library implementations, such as SevenZip or RarFile can't unpack empty folders and files (archives with CMS have a lot of them). Also users always like to see the execution process, which is impossible if the library doesn't allow (for instance if it just uses the extractall() method).

Solution:

The libraries ZipFile and libarchive-python had to be edited and connected as separate packages to the project. We had to create a library fork for libarchive-python and adjust it to Python 3.

Creation of zero size files and folders (bug found in SevenZip and RarFile libraries) had to be organized in a separate loop in the very beginning after file names in the archive. We contacted the developers concerning all bugs. As soon as we'll find some time, we'll send them a pull request, seems like they're not planning on fixing it themselves.

Gzip editing of compressed files (for SQL dumps etc.) has been created separately, which was possible without kludges with a standard library.

The operation progress can be controlled via a watcher with the system call IN_CREATE, using the pyinotify library. It sure doesn't work very accurately (the watcher doesn't always react, especially if there's a high number of files, that's why there's the magical coefficient 1.5), but it does show something similar. Not a bad solution, keeping in mind that there's no way to tracing that without rewriting all archive libraries.

Code for unpacking and creating archives.

Code example with comments

Raising security requirements

Task:

To keep the user from getting access to the endpoint server.

Problems:

Everyone knows that a web hosting server can host many hundreds of websites and users at the same time. First versions of out product allowed workers to execute some operations with root-priveleges, which theoretically (maybe) could have allowed to get access to someone else's files and read something you shouldn't or break it.

Sadly we can't give concrete examples. There were some bugs, but they didn't really have any influence on the server and also rather were our errors than a security hole. Anyhow, the web hosting infrastructure provides tools for load decreasing and monitoring, in the open-source version however we decided to seriously improve security.

Solution:

All operations had been extracted to so-called workers (createFile, extractArchive, findText) etc. Before getting started each worker performs a PAM authentication and a setuid user.

Whereby all workers work each separately in its own process and vary in its wrap (waiting or not waiting for response). That's why, even if the algorithm of one operation might be fragile, there's an isolation on the level of system permissions.

Neither does the application architecture allow to receive direct access to the file system, e.g. via the web server. This solution allows it to control the load quite effectively and monitor user activity on the server by any external means.

Setup

We chose the path of least resistance and prepared Docker images instead of a manual setup. Basically the setup is performed by a couple of commands:

```
user@host:~$ wget https://raw.githubusercontent.com/LTD-Beget/sprutio/master/run.sh
user@host:~$ chmod +x run.sh
user@host:~$ ./run.sh
```

run.sh will check the availability of images, download them in case of absence and start 5 containers with system components. To update images, execute

```
user@host:~$ ./run.sh pull
```

Image stop and deletion can be performed via stop and rm parameters respectively. You'll find a packaging dockerfile in the project code, the packaging takes about 10-20 minutes.

Shortly we'll write about setting up an environment for development on our website and in our Wiki on GitHub.

Help us improve Sprut.IO

There are quite a few obvious options for further improvement of this file manager.

We think most beneficial for the user would be:

Add SSH/SFTP support

Add WebDav support

Add a terminal

Add option for work with Git

Add file sharing option

Add theme changing, configuration and creation of various themes

Create a universal interface for work with modules

If you have thoughts on what might be useful for the user, share them in the comments or in the newsletter list sprutio-ru@groups.google.com.

We'll begin to implement them, but I'll just say it: doing it alone we'll spend years if not decades improving it. So if you [want to learn] can program, know Python and ExtJS and want to gain the experience of developing in an open project, we'd like to invite you to joint the development of Sprut.IO. Even more because every implemented feature will be rewarded by us since we won't have to implement it ourselves.

View the To-do list and task statuses on the project website in the section TODO.

Thanks for your attention! If you like, we'll be pleased to write more details about the project realization and will respond to your questions in the comments.

Project website: https://sprut.io

Demo version available at: https://demo.sprut.io:9443

Source code: https://github.com/LTD-Beget/sprutio

Russian newsletter list: sprutio-ru@groups.google.com

English newsletter list: sprutio@groups.google.com