Google Summer of Code 2016: Project Proposal for Drupal

PROJECT INFORMATION:

Treating this project as a real proposal, provide your implementation plan with as much detail as possible such as weekly time breakdowns, methods of mentor communication, project management, and when to expect specific results/deliverables.

Which project idea sparks your interest and why?

The project idea # 30 “Add Password-based Public Key Encryption for Drupal 8” sparks my interest the most. That’s because it is related to web security, something that I’m genuinely very concerned of. In fact, I wrote a lengthy article a few months ago on securely storing web passwords which demonstrates my interest in this area (reference: http://www.talhaparacha.com/web-passwords-database-security/).

Secondly, the idea aims to add a security feature which is very unique in the sense that no other CMS/CMF has it. This makes the project very exciting for me.

Lastly, the idea is based on ownCloud’s data encryption model. ownCloud is a free and open-source suite of client-server software, also written in PHP, for creating file hosting services. By successfully integrating their mechanism in Drupal, we’d present an excellent example of how open-sourcing a technology can help us all build great software.

Detailed Description:

(Available at https://groups.drupal.org/node/508466#project30 but a detailed description is still required to completely understand this because it’s a new idea and was not present in the original ideas list)

The goal of this project is to provide a role-based data encryption feature for securely storing highly sensitive data in Drupal 8. Accordingly, ownCloud’s Data Encryption Model will be used in this project which is proven to be truly secure and highly scalable. An end-user will just enable this new encryption mechanism for the sensitive fields in his website. And then even in the case of a full database breach, he’ll have the surety that his sensitive data is still well protected.

Problems with the Existing System:

The “Field Encrypt” module is used to encrypt fields before storing them in the database. The key used for encryption is then either stored in the database or placed at some directory other than the root for Drupal or moved to a remote server. Currently, only these three ways are available for key storage. But all of these ways are problematic because:

- In the first two cases where the key remains on the server, our data still remains very insecure though it's encrypted. That's because an attacker with a read access privilege to one part of the server should be assumed to have the same read access for any part of the server. Hence he can easily find the key used for encryption. This is figuratively equivalent to "storing the keys of your house under your welcome mat". This is in accordance with the views of other community members as one can see here: http://stackoverflow.com/questions/5834842/storing-sensitive-data-with-drupal/24395031#24395031
- In the third case where the key is moved to an external server, the website administrator now needs to subscribe to a Key Management service, quite unnecessarily. And if he chooses to manage the external server on his own, he'd get himself involved in much difficulty.
- Trusting the Key Management service to be secure and paying for it are other important issues. For example, Lockr charges $5 per month for only 5k key accesses.

Due to issues like these, the Encrypt module maintainers have noted on the project page that "...there are many levels that need to be thought about to have fully secure data".

Proposed Solution:

1. **User-keys:** We will generate a Public/Private key pair for each user. The Public-key of any user will get stored as it is in the database. But the Private-key of any user will get encrypted first with that user's login password and will then get stored in the database. In ownCloud's terminology, these are the User-keys.

2. **Role-keys/File-keys:** We will generate a key for each role present in the website. In ownCloud's terminology, these are the File-keys. But it makes more sense in our project to call them Role-keys because we are generating these keys for every role, not for every file.

3. **Share-keys:** Each Role-key will further get encrypted with appropriate users' Public-keys. Appropriate users mean all users who belong to that specific role. In this way, for one Role-key we will generate multiple keys all of whom will actually get stored in the database. In ownCloud's terminology, these encrypted copies of a single Role-key are the Share-keys.

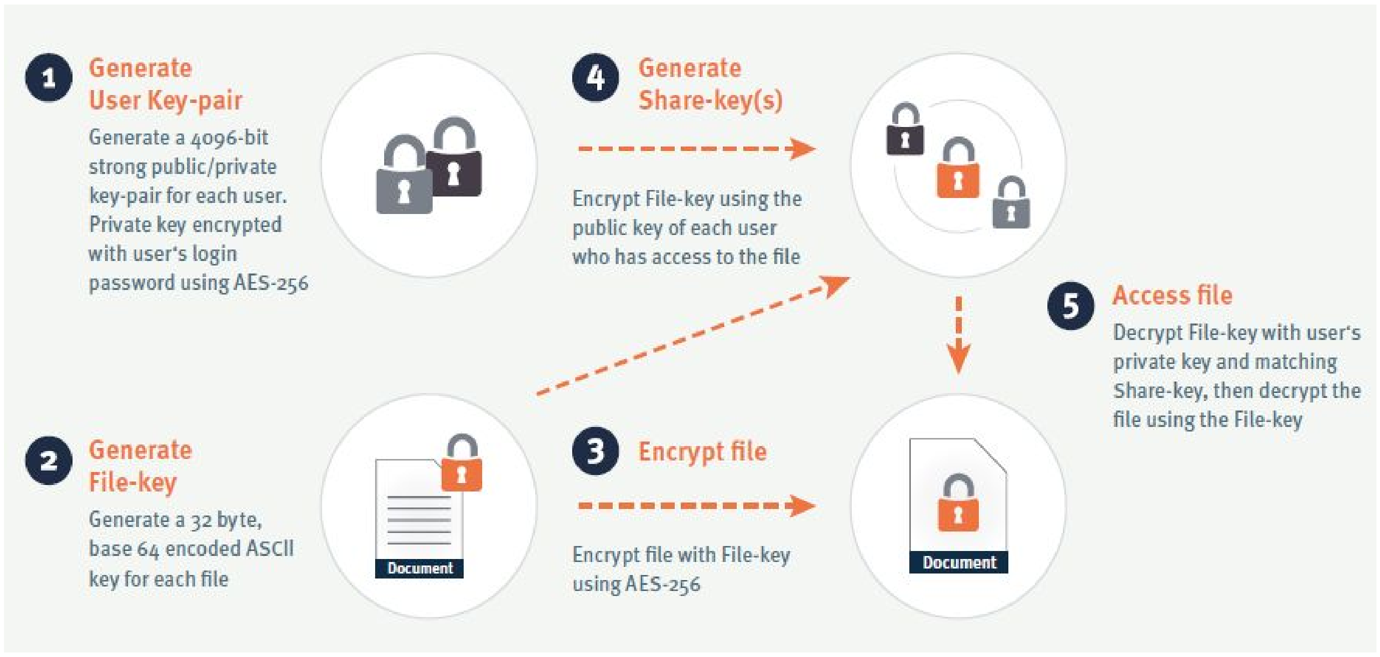
4. Whenever we want to securely store a piece of content, we will specify a Role-key which would be used to encrypt the content.

5. Whenever we want to access that piece of content, we will decrypt the Role-key assigned to that content through a combination of an authorized user's Private-key and matching Share-key. We will then use the Role-key to actually decrypt the content and return it to the user.

In this way the Role-keys, which will be used to actually encrypt the data, will not get stored anywhere on the server in their actual form.

6. Whenever a new user is added to a role, we will re-encrypt the Role-key for that role with this new user's Public-key to create a new Share-key. This will eliminate the expensive task of re-encrypting the entire content again and again whenever a new user is given access to some content. The same benefit is achieved when a user is removed from a role.

Though it's not necessary but to gain a very thorough understanding of this project, I'd recommend you to read ownCloud's paper on their data encryption model here: https://owncloud.com/owncloud-encryption-model. Or have a look at their illustration below which summarizes this mechanism perfectly:



1 Generate User Key-pair
Generate a 4096-bit strong public/private key-pair for each user. Private key encrypted with user's login password using AES-256
4 Generate Share-key(s)
Encrypt File-key using the public key of each user who has access to the file
5 Access file
Decrypt File-key with user's private key and matching Share-key, then decrypt the file using the File-key
2 Generate File-key
Generate a 32 byte, base 64 encoded ASCII key for each file
Document
3 Encrypt file
Encrypt file with File-key using AES-256
Document

Difference Between Ours & ownCloud's Implementation:

The previous section presented a summary on how the mechanism will work in Drupal's context. But ownCloud's implementation actually differs a bit. The difference is necessary to have a sustainable solution that we can easily implement by using D8's Key & Encrypt APIs. One can skip this section if he's not interested in finding out about this difference.

Accordingly, they generate a unique key for encrypting each file. But on the contrary, we will generate a unique key for each role. Hence in their case it's not possible that two files could get encrypted with a same key. But in our case it is possible for two fields to get encrypted with the same key. This will indeed happen if any two fields use the same Role-key for encryption.

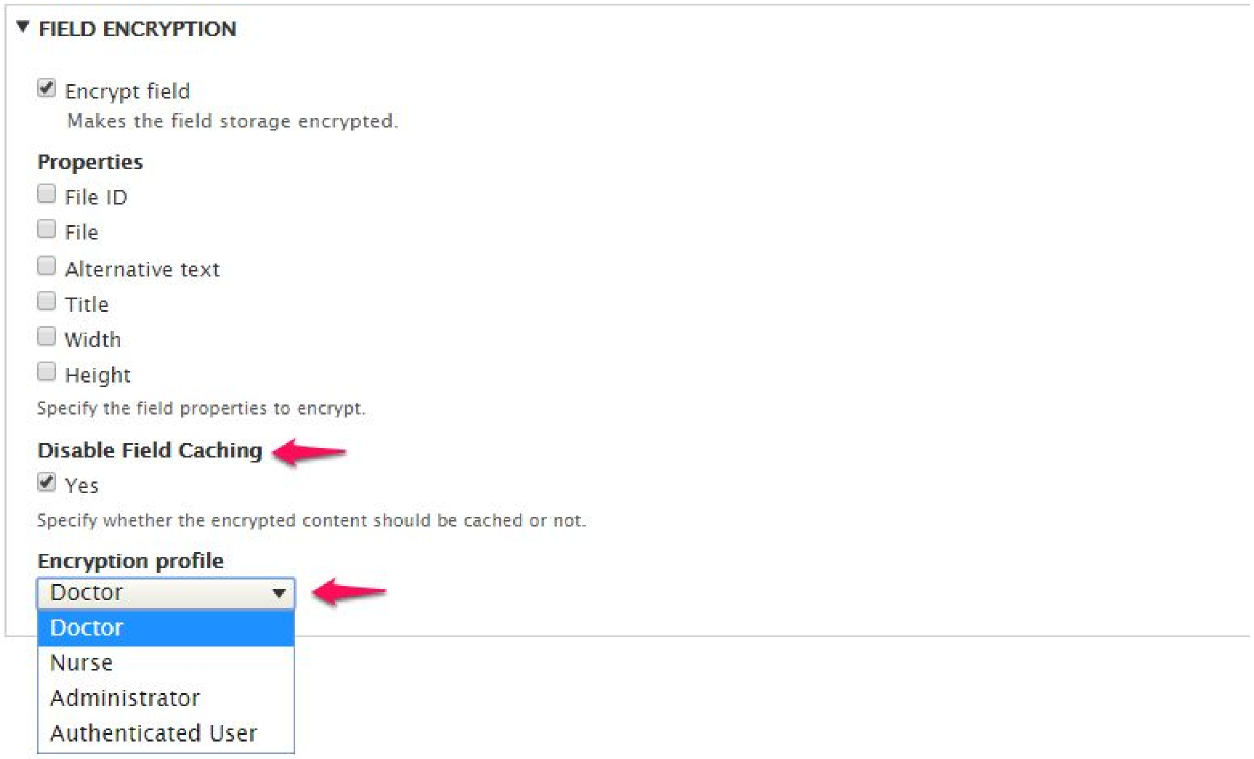
So, they have many File-keys, one for each file. We will have a few Role-keys, one for each role.

Example Use-case:

As an example, take the scenario of a Medical Records website where this feature could be of extreme benefit. Assume that the system stores patient records; doctors and/or nurses have the relevant permissions to view these records. Because privacy is one of the major concerns for such a system, any leakage of data would prove to be really hurtful for the organization storing these records. But with our mechanism in place, even in case of a full database breach no data leakage would occur.

How the Mechanism will Work from an End-user's Perspective:

The website administrator would simply enable our module along with its dependencies and will then go to the Storage settings for any sensitive field. The options this project will add there are denoted in this dummy screenshot below:



▼ FIELD ENCRYPTION

☑ Encrypt field
Makes the field storage encrypted.

Properties
☐ File ID
☐ File
☐ Alternative text
☐ Title
☐ Width
☐ Height
Specify the field properties to encrypt.

Disable Field Caching
☑ Yes
Specify whether the encrypted content should be cached or not.

Encryption profile
Doctor

So our module will generate an "Encryption profile" for each role present in the website. Each Encryption Profile would refer to some role and each role, in-turn, will be associated with a Role-key. Continuing with our previous example, suppose there is a field to store a patient's allergies and only doctors have the permissions to view it (doctors here refer to users of the website with the "Doctor" role). So the administrator would simply chose the corresponding Encryption profile for this field i.e. Doctor. Because this field stores sensitive information, the administrator would leave "Disable Field Caching" option checked. And that's all he'd ever have to do for securely storing that field!

And if the organizational policies are such that the nurses can view medical records too, the administrator can choose the "Authenticated User" encryption profile. Or the administrator can make a custom role, add appropriate users into it and then use the encryption profile corresponding to that newly generated role.

Actually, now the only way an attacker can access the data is if he has any doctor's password. But if he has that password then why not, instead of hacking the website and dealing with decryption, he just logs in to the website with that password and simply copies all the data he wants. In this way, the only exploitable point in our system will remain to be the passwords of users, but that has always been the case.

Expected Deliverables:

- Module handling Users' Public/Private Key pairs along with a few unit tests (by Week 3)
- Module providing the core functionality of this project (by Week 7)
- Several unit tests for the core functionality of this project (by Week 8)
- Performance benchmarks of the module (by Week 9)
- Patch for Field Encrypt module to deal with caching for encrypted fields (by Week 11)
- Documentation, video and a blog post about the module (by Week 12)
- Fully featured version of the module along with all unit tests (at the end-of-GSoC)

Weekly Breakdown with Implementation Details:

Community Bonding Period: Getting Familiar With Drupal 8's Architecture & Best Practices in OOP

- Drupalize.me will be providing all accepted GSoC students free access to their video tutorials. I plan to fully utilize this resource so to get myself very comfortable with development in Drupal 8 before the start of actual coding period.

Week 1: Managing Users' Public/Private Key Pairs

- Base fields for the User entity to store each user's Public/Private Key Pair will be created. Each pair will consist of two asymmetric 4096-bit keys.
- As explained earlier, the Private Key for each user won't be stored as it is, but would first get encrypted with that user's password and then this encrypted Private Key will be stored in the database. Hence a service for creating and storing Public/Private Key Pair for a user in this way will be defined. AES-256 Encryption would be used here.
- The aforementioned service will be hooked with the events of a new user registration, existing user's password change etc.
- A mechanism will be created for accommodating existing users' i.e. when an existing user will log-in with his password, the service for creating his Public/Private Key pair will be called if it hasn't been called before for that particular user.
- Upon a user log-in, the user's Private key will get decrypted with his password and will get temporarily stored in the user.private_tempstore. In this way, the Key API can easily access the Private key of the logged in user whenever it wants. It should be noted here that in case of a user's password change, his temporarily stored Private key in the user.private_tempstore won't get updated. That's because a password change would not imply a change in the Public/Private Key Pair for any user.

After looking at the codebase of ownCloud, I found out that they temporarily store the Private keys of their logged-in users in sessions. Therefore, we plan to utilize user.private_tempstore for this purpose. But this way works well with their use-case and conflicts with ours. This is because their use-case deals with two servers and they are going with the assumption that the server where ownCloud is installed, is safe but the server where data is kept, has been compromised. On the contrary, our use-case deals with a single server and we're going with the assumption that this main server has been compromised with illegal read-access privileges. Hence they can rely on sessions but we cannot.

Now there are a few other ways to deal with this. One is to use cookies for the temporary storage of a user's private key. The other is to not store the decrypted Private key temporarily but to always fetch it from the database; though this would require us to ask for a user's password every time his Private key is asked for. The former method adds an additional layer of security by bringing clients' machines into use while the latter provides bulletproof security but damages usability. Upon discussion with my mentors, we've decided to further explore this issue and ask for the opinions of other community members before making a final decision. In this regard, I've created an issue ticket here: https://www.drupal.org/node/2690753

Week 2: Testing Phase 1/3

Because our method of Password-based Public-key Encryption is meant to be highly-secure, a single error or bug can cause severe damage to an organization if that bug results in a data loss. Hence we need to extensively test our module and all of its use-cases.

Though I'm not opting for the test-driven development in its strictest sense, but I'll be writing the tests roughly every time after some new functionality is provided. To do this, I've planned to split testing into three phases; first of whom will start in this week.

- Unit tests for Public-Private key pair generation in case of a new user registration, in case of an existing user login and in case of a user's password change will be written.
- Any new bugs revealed during this testing phase will be resolved.

Week 3: Leveraging Drupal 8's Key API

- A "Key input" plugin will be implemented which will define our key as the combination of these two fields: the actual 32-byte key used to encrypt the data and some role defined in User-management.
- A "Key type" plugin will be implemented which will which handle the generation and validation of our newly defined key. It will also indicate that the "Key input plugin" used for this key type is the one we defined above.
- A "Key provider" plugin will be implemented which will handle the storage for our newly defined key type as well as other CRUD operations i.e. Getting the value of key by ID, Setting the value of key, Editing the value of key, Deleting a key etc.

Every time we want to store a key, our Key provider plugin will ensure that the actual encryption key doesn't get stored as it is but actually gets encrypted with the Public keys of corresponding users and then these multiple encrypted copies of the actual encryption key get stored in the database.

Every time we want to retrieve a key, we need to decrypt it's stored encrypted version with the Private Key of the asking user first and then return it. To do this, our Key provider plugin will access the user.private_tempstore where the corresponding Private key would be present.

Weeks 4 & 5: Leveraging Drupal 8's Encrypt API & Integrating with Field Encrypt Module

- An "Encryption method" plugin will be implemented which will define our encryption and decryption functions. Accordingly, AES-256 encryption will be used here too.
- For each role defined in the User Management, a key will be created of the Key type we defined earlier. And then using these keys, multiple Encryption Profiles will be created. So if, for example, a website has 3 Roles; Administrator, Authenticated User and Premium User, then 3 different Encryption profiles will be created each using one role as it's Key's second field.
- Whenever a user is given a role or is removed from a role, we need to re-generate the Encryption profile for that role. In this way the Key associated with that Encryption profile will be decrypted and retrieved and then encrypted again and stored. But this time, it will only get encrypted with the updated list of users' Public Keys thereby taking care of the fact that a new user has been added or removed from the role.

It should be noted here that Field Encrypt module hasn't been ported to Drupal 8 yet. But the work has already started and we will hopefully see its alpha-version by the mid of summer. And its architecture allows us to integrate our mechanism with it without worrying about the port.

In this way we'd integrate our mechanism with Field Encrypt module without ever touching a single line of its code. This is in accordance with the suggestions provided by the module maintainers of Key & Encrypt for implementing this project.

This indeed became possible due to the beautiful architecture of Drupal 8 where all the existing components are loosely-coupled and any new component can be integrated with much flexibility.

Week 6: Mid-term Evaluation

- Everything done so far will be reviewed and finalized for mid-term evaluation.

- No software project is completely certain. Hence to deal with any unexpected issues in implementation or to deal with any delays in the schedule from this proposed one, the time from this week will be utilized.

Week 7: Testing Phase 2/3

- Unit tests for Encryption Profiles creation/updation in case a new role gets generated and in case an existing user is added or removed from a role will be created.
- Unit tests for Fields Encryption and Decryption when any of our Encryption Profile is selected will be created.
- Any new bugs revealed during this testing phase will be resolved.

Week 8: Follow-up Performance Optimization & Benchmarking

ownCloud has utilized this Password-based encryption mechanism in both their Data Encryption Models 1.0 & 2.0 and they've been using it for quite some time now. They've found it to be very scalable, even in the case of very large files. That's because at the end of the day, we're only encrypting/decrypting small keys though we could be doing that a lot of times. Upon discussion with my mentors, we've reached to a similar conclusion that performance won't be a big issue but we realize that the only way to be sure of this is to test our hypothesis when we have the module working. Hence:

- Benchmarks will be created to illustrate how much our module slows things down. For example; a benchmark to note what time it takes for a field to get stored, provided that the role in the encryption profile used has 50k users (or 25k, 10k and so on). Subsequently, another benchmark to note how much time our module takes when a single more user is added to that role.

Claiming about a scalable solution is one thing but demonstrating it to be one is another; our benchmarks will play a vital role in convincing big organizations to use our mechanism as it will provide security without compromising much efficiency.

- All code written will be analyzed for fine-tuning and further optimization.
- Workarounds will be created if our benchmarks show a major performance bottleneck. One workaround is to create multiple encryption profiles for a role if it has more than a specific number of users. So if a role has around 50k users, for example, one encryption profile can cater for the first 25k users and the second for the rest. In this way, the total load for one role will get resolved by the two encryption profiles.

Week 9 & 10: Dealing with Caching

- Support for enabling/disabling caching for a field which has enabled encryption will be added.

- Render cache is the most relevant caching sub-system in play here and hence it will be tackled first.
- Other caching systems will be explored to see their role in a field’s storage/retrieval and will be tackled accordingly.

The current maintainers of Field Encrypt, Adam Bergstein & Rick Hawkins, have already thought of this feature for their module. But development for this piece hasn’t been started yet. So I conveyed my interest to them in developing this feature and they welcome my future contributions (though they want the feature to get developed early and thus if I get selected for GSoC, I might need to re-affirm with them the exact time when I’ll start development for this). In any case, this feature will be actually added to Field Encrypt module via a patch that I plan to create as a part of this project.

Week 11: Adding Custom Configurations Support & Testing Phase 3/3

Default configuration of the module is the one used in ownCloud’s own implementation. But different organizations follow different security standards. Hence to provide a bit of flexibility, support for custom configurations will be added in the module at this point.

- Support for setting custom key sizes (defaults are 32-bytes for Encryption and 4096-bits for Public/Private Key Pairs) and custom encryption method (default is AES-256) used in the module will be added via Drupal 8’s new configuration system.
- Few new test-cases will be added because of the new functionality provided.
- Any new bugs revealed during this final testing phase will be resolved.

Week 12: Providing Documentation etc. for the Module

- Because more time in software engineering is spent in maintaining a project rather than in building it, extensive documentation will be provided about the module as maintenance support. High-level block diagrams will be created to illustrate the overall architecture of the module. And a readme file to accompany the module will also be provided.
- A blog post will be drafted explaining this concept of encryption, the benefits of our module and its easy-of-use. I will try to get that post published on some popular Drupal or web-related blog. Otherwise I’ll host it on my personal blog.
- A video tutorial on installing and using the module will be created for site builders.

Week 13: Final Evaluation

- All expected deliverables will be finalized and submitted for final evaluation.
- A blog post summarizing my experiences with GSoC & Drupal will be written.

Mentors:

- Colan Schwartz, Enterprise Web Architect (https://www.drupal.org/u/colan)
- Jibran Ijaz, Drupal Developer at PreviousNext (https://www.drupal.org/u/jibran)

- Adam Bergstein, One of the Module Maintainers of Key & Encrypt (https://www.drupal.org/u/nerdstein)

Backup mentor: Christian López, GSoC Participant in 2007 & GSoC Mentor for Drupal in 2014 (https://www.drupal.org/u/penyaskito)

Project Difficulty:
Intermediate

Which aspect of the project do you see as the most difficult?
I see the aspect of dealing with caching as the most difficult part of the project. This is because there are several caching subsystems at play. And the Caching System has undergone a major architectural change in Drupal 8 and is brand-new so we cannot expect much documentation either. Hence I'd need to explore much on my own. As Adam Bergstein has told me, this whole caching aspect might be big enough to create a viable GSoC project of its own. So I realize that I won't not be able to get everything done. But I definitely plan to tackle the render cache subsystem which is the most relevant here.

Which aspect of the project do you see as the easiest?
I see the aspect of integrating this mechanism of Password-based Encryption with Encrypt module as the easiest. This is because we only need to implement the encryption/decryption methods and then bundle those with multiple encryption profiles. Encrypt already provides support for AES-256 encryption so we can use that. And the multiple profiles mentioned will be using the keys provided by the Key API. So this aspect is all about bundling a few things together.

Which portion of the project will you start with?
I'll stick with the schedule provided above and will start with the portion of managing users' Public/Private key pairs. This is the most logical way to start this project as these asymmetric keys form the base for our Password-based encryption mechanism. Handling this portion first will give me a solid foundation to build the rest of project upon.

How often will you communicate with mentor? How will you communicate with mentor?
Upon discussion with my mentors, we've decided that we'll have two 25-minute Skype meetings every week to discuss my progress on the project. The days we've decided for the meetings are Monday & Thursday and the timeslot we've decided is 11 PM Pakistan Standard Time. For regular discussions, we'll chat on Skype as I'll always be online there during my work hours.

We chose Skype because we've been using it for quite some time now and find it really effective. Through it, we can send messages to each other and answer them whenever we find ourselves convenient, but without losing their record. Plus we can share screenshots, PDFs and other media files relatively easily on it. And we always have the option of a video chat at our disposal if there's a need for it.

How will deal with project, task, and time management? Will you utilize software? If so, which tool and why?

Upon discussion with my mentors, we've decided to use Drupal.org for project and task management. So we'll have a public project where anyone can follow. I'll regularly add small tasks on the issues queue and will update & comment on their status as they get done. In this way, my mentors can track my day-to-day work activities and see for themselves whether I'm making progress on the project or not.

For time management, I've decided upon a schedule which I'll strictly adhere to during GSoC, if I get selected; I will work from 1 PM to 4 PM and from 11 PM to 4 AM every weekday.

Future Work (to be done after finishing GSoC):

- I plan to remain the maintainer of the module I'll develop as a part of this project and will provide support for bug fixes whenever necessary.
- The goals of this project, as originally crafted by my mentor, contained Integration with File Encrypt. But there's absolutely no work done on porting File Encrypt to Drupal 8. It will probably have an architecture similar to that of Field Encrypt and hence when its port gets done, integration with our module won't take any time. So I plan to do this after the completion of this project because by that time we'll surely see some development on File Encrypt.
- The goals of this project, as originally crafted by my mentor, contained Support for Views. Field Encrypt already provides this functionality so there's no need to work on that. But views' filters don't work with any field which has encryption turned on. This is because filters send direct queries to the database where content might not be stored in its actual form but rather in an encrypted one. Hence whenever an administrator tries to use such a field in a filter, the module Field Encrypt should warn him. I plan to add support for this feature. This is in accordance with the issue: https://www.drupal.org/node/2155339.

About Me:

I'm a 2nd year Software Engineering student from Pakistan and am in love with everything related to tech & computing. I started programming in 8th grade and have been doing it passionately since then. Last year, I won the Women's Transport Innovation Hackathon sponsored by UN-Women (https://twitter.com/unwomenasia/status/644713185577201664) for building an SOS app for women's safety. In the same year, I also won the SEECS Social Hackathon sponsored by Telenor (https://goo.gl/YWDvvz) for building a college entry-test preparation app. Apart from that, I participated in ACM ICPC Lahore Regionals 2015 so I also have a bit of experience in competitive programming.

Terms & Conditions:

I agree to attend the weekly check-in meeting and understand that missing two meetings will result in a failure.

Thank you for reading till end. Please reach out to me if there are any further queries.