Advanced Enterprise Computing

Blockstarter 4.0 Prototyping Project Report

Group E

Ahmad Jawid Jamiulahmadi
Aqa Mustafa Akhlaqi
Filippo Boiani
Gabriel Vilen
Hekmatullah Sajid
Riccardo Sibani
Rohullah Ayobi
Stefan Stojkovski

GitLab repo

https://gitlab.tubit.tu-berlin.de/stefan.stojkovski/aec-final-project

Introduction

The distributed database blockchain is drastically increasing popularity in software development and many other fields. In this report we present Blockstarter 4.0, a kickstarter-inspired project built upon the blockchain for the course *Advanced Enterprise Computing* at TU-Berlin, summer semester 2017. Blockstarter 4.0 consists of two parts, first one of a backend and smart contracts, and the second part (*extension*) handles projects with deadlines.

The first part, *backend and smart contracts*, has the following requirements:

- allows project owners to withdraw funds when a project has been funded successfully (and not be able to withdraw if the funding goal has not been reached, yet)
- allows backers to retrieve a share (token) for each project they invest in

In the second part we choose *Projects with deadline*:

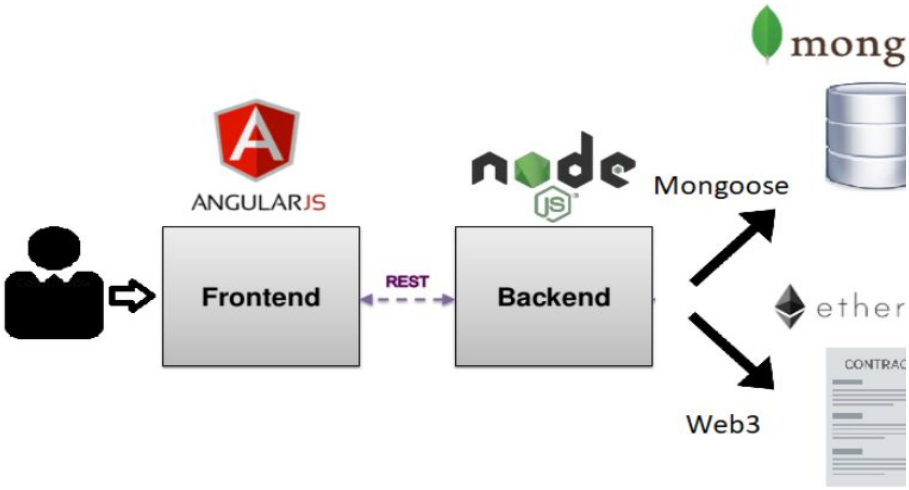
- A project has a deadline until which the funding goal must be reached. Otherwise, the project is automatically cancelled and all backers are refunded.

Concept and design

For the concept and design we have presented both teams approaches and solutions to exercise 5 and have decided to reuse parts of the code of both teams and upgrade the application according to our requirements.
We have also decided to implement the extension in which project has a deadline until which the funding goal must be reached.
In order to achieve our implementation goals we have decided to use AngularJS for the frontend, and NodeJS connecting to MongoDB and to the Ethereum testrpc blockchain.
In terms of project structure and architecture we have decided to go with the following architecture:

- Frontend build with AngularJS
- Contracts deployed on Etherium testrpc
    - Contract for Project deployment and management
    - Contract for Token issuing
- Additional Document database (MongoDB) as helper for listing all projects and project per backer and user management

    - We have decided to choose MongoDB because of the easy integration with Node.js through mongoose, and the document structure perfectly fits our requirement, with easy querying and less cost and overhead than implementing this requirement directly in the contracts because of the expensive writes.
- Backend written in Node.js which exposes APIs to the frontend to retrieve and display relevant data by communicating with the contracts or database according to the request type.



Before reaching agreement for the final architectural implementation, we have discussed few approaches for token issuing:

- Return token along with the TX address after investing
    - Eliminated because we can't determine the proper ratio in advance if the project gets more money than the project goal and if the project doesn't reach the funding goal.
- Distribute tokens automatically after the funding goal is reached
    - Eliminated because there is not explicit notification to the backers that the goal is reached and that the token is distributed to them
- Allow the backers to claim the token only after the funding goal has been reached
    - Chosen approach because it is easiest to determine the right ratio and value for the money each backer has invested

Another discussion we had was regarding implementation of the extension for implementing project deadline and have identified 3 possible approaches:

- Implementing another scheduling contract which executes automatically and kills the contract if the funding goal has not been reached after certain number of blocks have been written into the blockchain
    - Eliminated because this contract scheduler can be implemented in the Geth Ethereum Testnet only and not in the testrpc on which we develop and build our project.

- Implementing Event emitter in the backend with Node.js which checks if the funding goal has not been reached and then calls the function to kill the contract
    - Not the most elegant, but easiest to implement and without other implications
- Implementing the checks as function modifiers on each of the functions called which checks the deadline and the funding goal and kills the contract if the goal has not been reached after the deadline
    - The implications of this approach are that the clients which have invoked the functions have to pay ether for killing of the contract and the contract can live after its deadline has passed if no function has been called to check the contract status

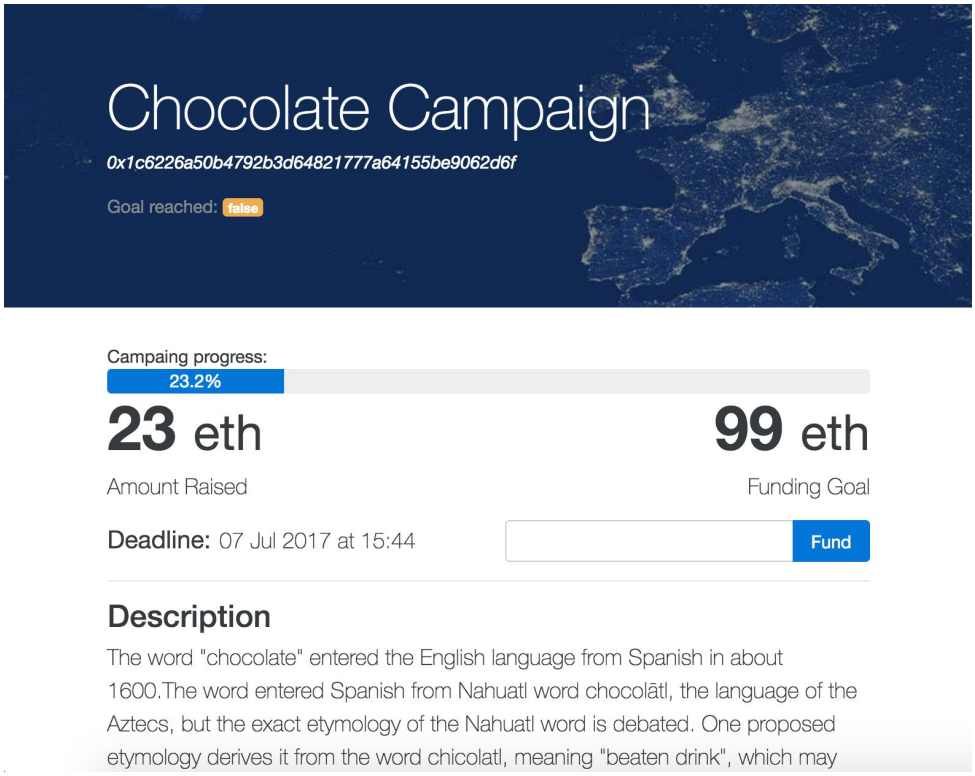
Implementation

Frontend

The frontend requirements for Blockstarter 4.0 were:

- Allows users (blockchain address) to create new projects
- Allows project owners to list their projects with funding status
- Allows project owners to cancel a project and refund all backers.
- Allows backers to invest in a project
- Allows backers to view in which projects they have invested

We started off building the frontend app with Meteor, a javascript library for quickly building and deploying apps. However, we had problems integrating it with the apis, so because of lack of time we switched to angular.js and react. Below is a printscreen of the final frontend of a project called “Chocolate Campaign”.



The frontend app consist of five files, api, app, auth, config and controller which are highlighted below.

Api.js

The api.js file uses $http, a core angular service which facilitates communication with a HTTP server via the browser's XMLHTTPRequest or JSONP. Our functions returns a promise containing the http response. For example, addProject is called when a user wants to add a new project. It takes the new project as a argument and sends a POST to the endpoint path specified in the config file plus “/projects” and return the response data.

```
this.addProject = function(project) {
    return $http
        .post(CONFIG.endpoint + '/projects', project)
        .then(response => { return response.data; })
        .catch(error => { return error; });
};
```

App.js

app.js runs when the app start and set ups and configures the router. This is done in the .config function which specifies the template and controller to use.

```
.config(function($routeProvider, $httpProvider) {

    $routeProvider
        .when('/projects/view', {
            templateUrl: 'templates/projects.html',
            controller: 'ProjectsCtrl'
        })
```

The app.js also contains functions for handling the login functionality, which is done by specifying a testrpc address.

Auth.js

This is the authorization module and contains services for the authorization. We have a local (key, value) storage which stores the user credentials, currently address and associated token key, and is used to check if the user is authorized. Currently we're using a fake login since the login functionality wasn't specified in the requirements.

```
/**
 * AUTHENTICATION SERVICES using JWT + LOCALSTORAGE
 **/
.service('AuthService', function($q, $http, CONFIG, AUTH_EVENTS, $loc
    let LOCAL_TOKEN_KEY = 'BlockstarterAuth';
    var isAuthenticated = false;
    console.log("AuthService: " + isAuthenticated);

    function loadUserCredentials() {
        let token = $localStorage.getObject(LOCAL_TOKEN_KEY);
        if (token)
            useCredentials(token);
    }

    function storeUserCredentials(token) {
        $localStorage.setObject(LOCAL_TOKEN_KEY, token);
        useCredentials(token);
    }
```

Config.js

The config file basically configures the modules by containing constant path and abbreviation.

```
angular.module("Blockstarter.config", [])
    .constant('AUTH_EVENTS', {
        notAuthenticated: 'auth-not-authenticated',
        notAuthorized: 'auth-not-authorized'
    })
    .constant("CONFIG", {
        "appVersion": "1.5.0",
        "base_url": "http://localhost:8080",
        "endpoint": "/api/v1",
```

Controller.js

There are a bunch of different controllers (AppCtrl, ProjectsCtrl, CreatorsCtrl, BackersCtrl, LoginCtrl, UserCtrl) that handles the interaction between different component in the app. For

example, the fundProject takes a project and calls the api methods for adding and decreasing on it.

```
$scope.fundProject = (project, amount) => {
    const req = {
        project,
        amount,
        backer: user.address
    };
    console.log(req);
    Api
        .backProject(req)
        .then(response => {
            console.log(response);
            //window.location.reload();
            $scope.project.fundingStatus = parseFloat($scope.project.fundingStatus) + parseFloat(amount);
            $scope.project.finalFundings = parseFloat($scope.project.finalFundings) + parseFloat(amount);
            $scope.project.goalReached = $scope.project.finalFundings >= $scope.project.fundingGoal;
        })
        .catch(error => console.error(error));
};
```

Backend implementation

The backend implementation is written in Node.js using Express framework for creating the server APIs and Web3 in order to interact with the Ethereum testrpc blockchain.
The main features implemented can be found in the src folder, the contracts can be found in contract folder and the models in the models folder.

Contracts:

ShareToken.sol
Before each project is created, we need to create a token contract where we specify the initial supply of tokens available to the backers.
In the beginning the creator of the contract is in the possession of available tokens. When the funding goal is reached and when the backers request to claim the token shares, the appropriate amount of shares is transferred from the project(token) creator to the backer. This information is kept by mapping the address of the user to uint256 as an array with all the balances.

```
contract ProjectShare {
    string public name;
    string public symbol;
    uint8 public decimals;

    /* This creates an array with all balances */
    mapping (address => uint256) public balanceOf;

    /* This generates a public event on the blockchain that will notify clients */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /* Initializes contract with initial supply tokens to the creator of the contract */
    function ProjectShare(uint256 _initialSupply, string _tokenName, uint8 _decimalUnits, string _tokenSymbol) {
        if (_initialSupply == 0) _initialSupply = 100;
        balanceOf[msg.sender] = _initialSupply;    // Give the creator all initial tokens
        name = _tokenName;                                   // Set the name for display purposes
        symbol = _tokenSymbol;                               // Set the symbol for display purposes
        decimals = _decimalUnits;                            // Amount of decimals for display purposes
    }

    /* Send coins */
    function transfer(address _to, uint256 _value) {
        if (balanceOf[msg.sender] < _value) throw;           // Check if the sender has enough
        if (balanceOf[_to] + _value < balanceOf[_to]) throw; // Check for overflows
        balanceOf[msg.sender] -= _value;                     // Subtract from the sender
        balanceOf[_to] += _value;                            // Add the same to the recipient
        Transfer(msg.sender, _to, _value);
    }
}
```

Project.sol

The following functionalities are implemented in this contract

- Receive Ether from anonymous Backers
- Update the funding status of the project
- Show funding status of the project
- Kill the project and redistribute the funds (only Creator)
- Withdraw funds (only Creator)
- Retrieve a share (only Backer)

We are using modifiers in order to restrict the access to the functions to certain callers only as well as to assert if the campaign is open or closed by inspecting if the goal is reached.
We are using events in order to keep track and give feedback to the backend when someone has backed a project, a share has been claimed or funds have been withdrawn.
Other things worth noting are:

- We use mapping (address=>Backer) in order to keep track of the amount backed by a Backer and if the Backer has claimed his shares
- We use helper iterator array where we push the addresses of the backers in order to be able to iterate and pay back the funds if the contract is killed

app.js

Here is all the initialization and startup setup including:

- import of the 3rd party dependencies
- setting the application environment variables
- initializing MongoDB and web3
- Executing script to start testrpc
- Starting the application to listen on specified port

Database.js

Helper for unified access and querying to the Mongodb. We pass the model in the constructor and according to the model which is either backer or creator we can query the database to get the required data

```
module.exports = class Database {

    constructor(model) {
        this.Model = model;
    }

    getAll() {
        return this.Model.find();
    }

    update(data, query) {
        return this.Model.findByIdAndUpdate(data, query, { safe: true, upsert: true });
    }

    getProjectsByAddress(query) {
        return this.Model.find(query).select('projects -_id');
    }


    updatePull(query) {

        return this.Model.update({}, query, false, false);
    }

}
```

The deployment and the interactions with the contracts are written in deploy.js and interaction.js accordingly.

deploy.js

The script is written to deploy the two types of contracts we have using web3. Token creation contract is deployed with the createToken function to which we pass initial supply, token name, decimals and token symbol to be used. Project creation contract deployed with the createProject function to which we pass the title, description, goal, price and the address of the token contract which must be created prior to the creation of the project contract. Both function are implemented with Promises because of the async. nature of node.js and the “long” duration of contract deployment.

Also we log the time it takes to deploy the contract with time.stop() function and we return this information in the results along with the creator wallet address(for project creation only), the address of the contract deployed and the instance of the contract.

```
let createProject = data => {

    return new Promise((resolve, reject) => {
        // start a timer
        time.start();
        console.log(`Start deploying project ${data.title}...`);

        // deploy the contract

        web3.eth
            .contract(projectContract.abi)
            .new(data.title, data.description, data.goal, data.price, data.token, data.duration, {
                from: data.creator,
                data: projectContract.unlinked_binary,
                gas: '4700000'
            }, function(error, contract) {

                if (error)
                    reject(error);

                if (typeof contract.address !== 'undefined') {
                    // TODO: change creator
                    var result = {
                        creator: data.creator,
                        address: contract.address,
                        time: time.stop(),
                        instance: contract
                    }
                    resolve(result)
                }
            });
    });
}
```

Interaction.js

Used to interact with the deployed contracts and call different contract functions that are available and needed using web3. For each function available in the contract we have defined equivalent function call in node.js using web3. To each function we pass the same data needed accordingly. For the functions that take longer to execute, we use Promises again.

The following functions are implemented:

- getProjectContract
- getTokenContract
- fundProject
- showStatus
- getProject
- setParams - set project parameters
- withdrawFunds
- claimShare
- Kill

The logic for checking if, for example, the owner can withdraw funds only if the goal is reached and if he is the owner of the contract is done in the solidity contract directly.

Another thing worth noting is that we have created events in the solidity contract in order to monitor the proper execution of the contract and to return valid information to the end user.

The events implemented are the following:

- Someone backed a project
- Funds are withdrawn from a project
- Shares are claimed by the backer

Example function to reference the described above:

```
let fundProject = data => {
    return new Promise((resolve, reject) => {
        // event: when someone fund a project it shows the address and the amount backed.
        let someoneBacked = getProjectContract(data).SomeoneBacked({ fromBlock: 0, toBlock: 'latest' });

        // fund the project
        web3.eth.sendTransaction({ from: data.backer, to: data.project, value: data.amount, gas: 500000 });

        // start watching for the funding event
        someoneBacked.watch((error, result) => {
            if (error) {
                someoneBacked.stopWatching();
                reject(error);
            } else {
                resolve(result);
            }
        });

    });
}
```

router.js

All the APIs are implemented here using express framework.

We use 3 controllers:

- Deployer - used for contract deployment
- Interactor - used for calling contract functions
- Scheduler - used for project deadline extension implementation

The flow for project creation is the following:

1. Deploy token contract for the project and assign how many shares are available to the backers
2. Deploy project contract including the address where the token contract is deployed
3. Update Creator mongodb by pushing the new project to the project array associated with the creator
4. Set the Scheduler to implement project deadline extension
5. Return result

```
app.post('/api/v1/projects', function(req, res, next) {
    logRequest(req);
    deployer
        .createToken(request.token)
        .then(result => {

            // set the token address inside the project
            request.project.token = result.address;
            console.log(`Token ${result.address} created in ${result.time} seconds`);
            // create the project and add a promise to the chain
            return deployer.createProject(request.project);
        })
        .then(result => {

            console.log(`Inserting ${request.project.title} into ${result.creator} projects list`);
            let deadline = new Date().setTime(new Date().getTime() + (request.project.duration * 60 * 1000));
            //write to mongoDB
            let query = {
                $push: {
                    "projects": {
                        address: result.address, // project contract address
                        token: request.project.token, // token contract address
                        deadline: deadline
                    }
                }
            }

            // TODO: change the promise: http://mongoosejs.com/docs/promises.html
            CreatorDB
                .update(result.creator, query)
                .then(result => console.log(`resultssss: ${result}`))
                .catch(error => console.log(error));

            let data = {
                creator: result.creator,
                project: result.address
            };
            var r = `Project ${request.project.title} with address ${result.address} created in ${result.time} seconds\n`;
            let s = scheduler.setScheduler({ deadline: deadline }, data);
            console.log("asdfasdf");
            res.send(r);

        })
        .catch(error => res.send(error))
        .catch(error => res.send(error));
});
```

The flow for funding a project is the following:

1. Call fund project function in the project contract
2. Update Backer mongodb by pushing the new project funded associated with the backer
3. Return result

```
app.post('/api/v1/projects/fund', function(req, res, next) {

    logRequest(req);

    let funding = req.body;

    interactor
        .fundProject(funding)
        .then(result => {
            let backer = req.body.backer;
            let query = { $push: { "projects": { address: funding.project } } };

            // write on db
            // TODO: change the promise: http://mongoosejs.com/docs/promises.html
            BackerDB
                .update(backer, query)
                .then(result => console.log(result))
                .catch(error => console.log(error));

            res.send(result);
        })
        .catch(error => {
            console.log(error);
            res.send(error);
        });
});
```

The flow for getting all projects created by creator is the following:

1. Query mongodb to get the list of project addresses
2. Create array of Promises to query the status of each project from the blockchain
3. Return result

```
// Get all projects created by a creator
app.get('/api/v1/projects/creator/:creator', function(req, res, next) {
    logRequest(req);
    let query = { _id: req.params.creator }
    CreatorDB
        .getProjectsByAddress(query)
        .then(result => {

            let projects = result[0].projects;

            let promisesArray = [];

            for (var index = 0; index < projects.length; index++) {

                let data = { project: projects[index].address };

                promisesArray[index] = interactor.showStatus(data);
            }

            Promise.all(promisesArray).then(result => {

                console.log(result);
                res.send(result);

            }).catch(error => console.log(error));
        })
        .catch(error => res.send(error));
});
```

The code of the other APIs is more simple and self explanatory.

The following APIs call are implemented:

Method	Endpoint	Params	Returns	Description
GET	/api/v1/		A hello greeting string from Group E	Get Hello World
POST	/api/v1/projects	-project -backer -amount		Create a project

POST	/api/v1/projects/fund	-project -backer -amount		Fund a project
GET	/api/v1/projects			Get all projects
GET	api/v1/projects/creator/:creator			Get all projects created by a creator
GET	/api/v1/projects/backer/:backer			Get all projects funded by a backer
GET	/api/v1/projects/status/:project			Show project status
GET	/api/v1/projects/:project			Show project information
POST	/api/v1/projects/withdraw	-project -creator -amount		Withdraw funds from the project
POST	/api/v1/projects/show/shares	-project -backer -token		Show project shares
POST	/api/v1/projects/claim-shares	-project -backer		Claim project shares

Example request body of **createProject**:

```
{
    token: {
        initialSupply: 10,
        tokenName: "Example Token",
        tokenSymbol: "Symbol",
        decimals: 4,
        creator: "0x ..."
    },
    project: {
        title: "Example",
        description: "Frist Project With Token",
        goal: 100,
        duration: 120, //minutes
        sharesAvailable: 50,
        creator: "0x ..."
    }
}
```

Scheduler.js

The project deadline extension is implemented here.

The implementation consists of the following:

1. For each project created a timeout is set and killContractIfFundingGoalNotReached function is called when timeout event occurs
2. The killContractIfFundingGoalNotReached function checks the status of the project by calling the showStatus function deployed in the project contract in the blockchain
3. If funding goal is not reached the contract is killed by calling the kill function deployed in the project contract

```
this.killContractIfFundingGoalNotReached = data => {
    interactor
        .showStatus(data)
        .then(result => {
            if (!result.goalReached) {

                let query = { $pull: { "projects": { address: data.project } } }
                    //remove the project from array of creator projects
                CreatorDB
                    .updatePull(query)
                    .then(result => console.log(result))
                    .catch(error => console.log(error));

                //remove project reference to backers
                BackerDB
                    .updatePull(query)
                    .then(result => console.log(result))
                    .catch(error => console.log(error));

                interactor.kill(data);
            }
        });
}

this.setScheduler = (project, data) => {
    let time = project.deadline - (new Date().getTime());
    console.log(time);
    let timerObj = setTimeout(this.killContractIfFundingGoalNotReached, time, data);
    return timerObj;
}
```

Database models

The database models are simple. We have simple schema. Creators with the associated project array for project created by them and we have backer with the associated project array for the projects they have backed.

Design Decisions

Selecting event-based approach over array structure with promise:

For the project we evaluated the code of exercise 5 of both groups and selected one as base for further implementation of the project. Most of the implementations were the same except that one group used event and the other group used array of structure for the *payable* function. The payable function is called when a backer wants to invest in a project, this function adds the invested amount to the contract account.

We selected the event-based implementation over the array of structure implementation. The reason for this is that with event we are waiting for the event completion, when it is completed we are sure that funding was successful. But in array of structure it was not possible to make sure that funding was successful or failed.

Technology Selection:
At first we selected Meteor framework with React library for frontend development. Unfortunately we got problems while integration of Meteor with API, the main problem with Meteor was that it wasn't working well with callbacks and promises. The mentioned problem with Meteor resulted to use AngularJS for the frontend development.
The most important good things about Meteor which some are listed here. [2,3]

- You are noramly building the web application and it is going to be real-time.
- You can develop with single language, the development process is simplified by combining the frontend, backend and database into a single language.
- Its packages allows to save lot of time.
- Optimized for developers with good community support.

The good things about ReactJS is that it is efficient. It It makes writing Javascript easier by using a special syntax called JSX, which allows us to mix HTML with Javascript. It gives you out-of-the-box developer tools like official React.js chrome extension which makes the debugging much easier.
We selected MongoDB for storing the project and owner information rather than PostgreSQL. The reason for this decision was that if we look into the relation of owner and project, every project has only one owner. In this scenario the MongoDB suits well, if we had more than one owner per project than using PostgreSQL was better option for storing the one to many relationship between owners and project. Flexibility of schema easy integration of MongoDB can be listed as additional positive points, and also it is part of main stack.
Extension:
The group decided to implement the first extension “Project with deadline”. For implementation of the extension we are using scheduler. When a new project is created and the deadline date and time is specified. A scheduler is created which runs until the deadline, while deadline is reached the funding goal is checked and acts accordingly. If the funding goal is not reached the project will cancel and all backers are refunded. We store the scheduling data in mongodb, if the application is closed, after running again all scheduling data is read from mongoDB and resume the scheduler for those projects that their deadline was not reached at the time of closing the application.

Project Management

Group Organization
There are two exercise groups are joined (Group R and Group S) for this project (Blockstarter 4.0 with Extensions).

Project Methodology
The development methodology for this project is *Scrum* which is an *Agile* methodology.

Scrum

We customized the Scrum for reaching our goals. We tried to have short Sprint iterations to use the time in a better way.
As *Slack* sufficed for internal group communications, we didn't feel the need to meet up daily for sprint executions but we had sprint meetings multiples time of the week.

The reasons behind using scrum were:

- we are a cross functional team with different capacities and skills
- we emphasised on face to face communication
- we had two different approaches for implementation; scrum helped us to respond easily for any changes during project implementation
- Multiple meetings during the week helped us to better evaluate the project development and issues

Pair programming
Beside scrum we have tried to learn some engineering practices associated with *XP* which was *Pair Programming*.

Here is how pair programming helped us reach our goals:

- Better code with higher quality design
- It helped us to quickly get to know each other in our team
- Pair programming helped us to easily identify the issues and bugs in code, it even helped us to evaluate different approaches for implementing our design concepts.

Meetings (Update Meetings)
We decided to use Agile (Scrum) methodology for the development of the project with one iteration per week.

- *Sprint Planning Meeting:* where we discussed re-implementation of the Blockstarter from the previous exercise groups.
- *Sprint Review Meetings:* where we demonstrated the working software versions and discussed improvement for different and new functionality and extensions.
- *Update Meetings:* where we met with the supervisor discussing improvement and issues on project progress.

Communication channels
Slack: we decided to use slack for internal group communication in order to easily separate our discussions into particular channels.
Slack WebHook: automation and notification for every commits for the *GitLab* into *Slack*.

Documentation Organization
It is necessary to have documentation for both the source code and the overall documentation. We decided that everything should be documented as we go, all (design) decisions should be documented as far as options that were considered and the weighing of pros & cons.

Source code documentation
Every source code details are commented inside the code itself. It's also has been decided to use GitLab for VCS.

- Version Control System (GitLab)

We found GitLab convenient so we started pushing all our source code into it. We used separate branches for particular section of our system such as *Frontend, API, Smart Contract* which we then merged them into master branch.

References

1. https://angularjs.org/
2. https://docs.angularjs.org/api/ng/service/$http