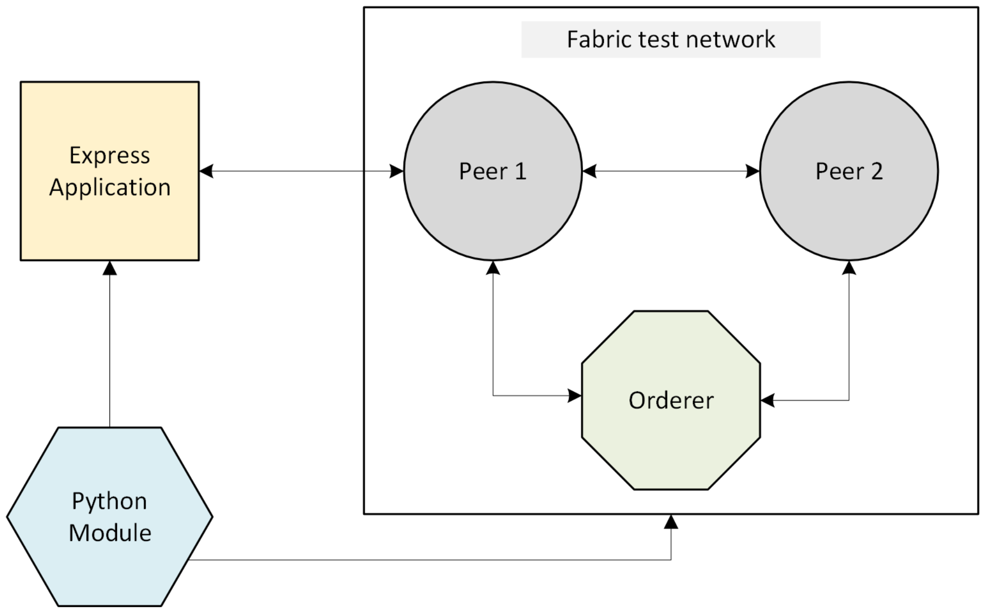
Introduction

In this assignment, we have developed a decentralized application for energy resource management and auction control using blockchain and smart contracts. The key technologies utilized in our implementation include:

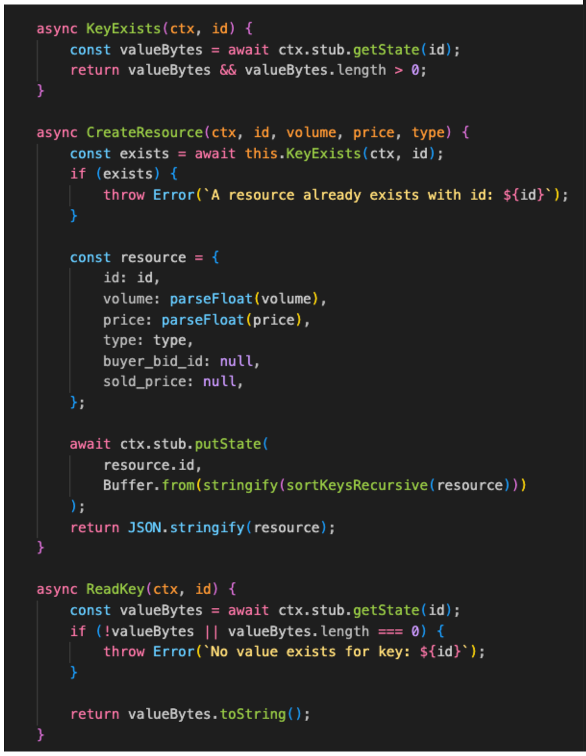
- **Hyperledger Fabric:** An enterprise-grade blockchain framework used to simulate the decentralized network. We developed chaincodes (smart contracts) within this framework to enforce auction control logic autonomously and securely.
- **Express.js Application:** Since Hyperledger Fabric operates within Docker containers, an external application is required to interact with network peers and invoke chaincode functions. To facilitate this, we developed an Express.js-based API that communicates with the blockchain network via GRPC connections, enabling seamless interaction between external modules and the decentralized system.
- **Python Module:** A Python script orchestrates the deployment and execution of the network components, including the Express.js application. Additionally, it automates the benchmarking of optimized and non-optimized smart contracts, ensuring efficient evaluation with a single command.

The following figure illustrates the interaction between these components in our system.

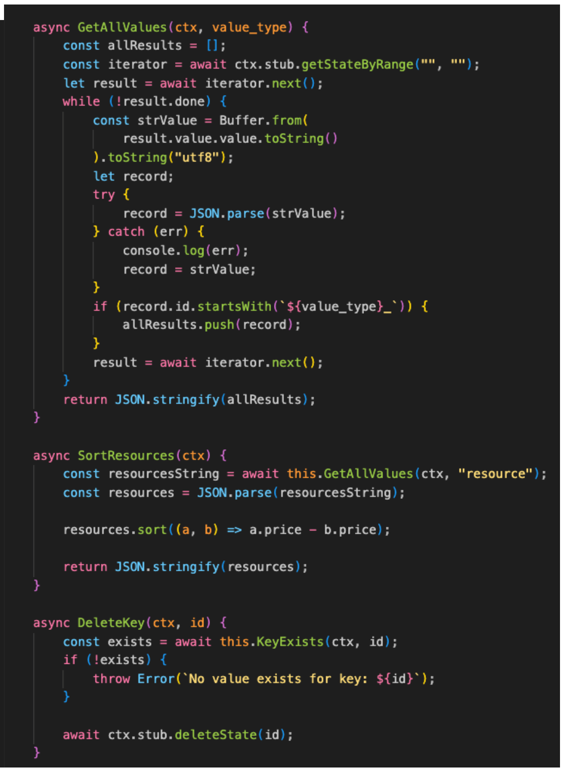


Smart Contract for Resource Management

In this section, we developed a smart contract to manage energy resources, implementing essential operations such as Create, Read, Update, and Delete (CRUD) functions. These functions ensure secure and transparent handling of resource data on the blockchain. The following figures showcase the implementation of these smart contract functions.

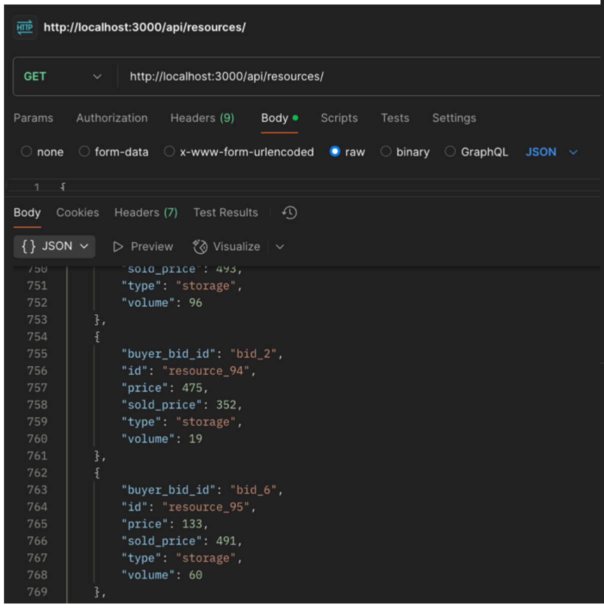


```javascript
async KeyExists(ctx, id) {
    const valueBytes = await ctx.stub.getState(id);
    return valueBytes && valueBytes.length > 0;
}

async CreateResource(ctx, id, volume, price, type) {
    const exists = await this.KeyExists(ctx, id);
    if (exists) {
        throw Error(`A resource already exists with id: ${id}`);
    }

    const resource = {
        id: id,
        volume: parseFloat(volume),
        price: parseFloat(price),
        type: type,
        buyer_bid_id: null,
        sold_price: null,
    };

    await ctx.stub.putState(
        resource.id,
        Buffer.from(stringify(sortKeysRecursive(resource)))
    );
    return JSON.stringify(resource);
}

async ReadKey(ctx, id) {
    const valueBytes = await ctx.stub.getState(id);
    if (!valueBytes || valueBytes.length === 0) {
        throw Error(`No value exists for key: ${id}`);
    }

    return valueBytes.toString();
}
```



```javascript
async GetAllValues(ctx, value_type) {
    const allResults = [];
    const iterator = await ctx.stub.getStateByRange("", "");
    let result = await iterator.next();
    while (!result.done) {
        const strValue = Buffer.from(
            result.value.value.toString()
        ).toString("utf8");
        let record;
        try {
            record = JSON.parse(strValue);
        } catch (err) {
            console.log(err);
            record = strValue;
        }
        if (record.id.startsWith(`${value_type}_`)) {
            allResults.push(record);
        }
        result = await iterator.next();
    }
    return JSON.stringify(allResults);
}

async SortResources(ctx) {
    const resourcesString = await this.GetAllValues(ctx, "resource");
    const resources = JSON.parse(resourcesString);

    resources.sort((a, b) => a.price - b.price);

    return JSON.stringify(resources);
}

async DeleteKey(ctx, id) {
    const exists = await this.KeyExists(ctx, id);
    if (!exists) {
        throw Error(`No value exists for key: ${id}`);
    }

    await ctx.stub.deleteState(id);
}
```

Here are the results of running these smart contracts using the Express APIs:



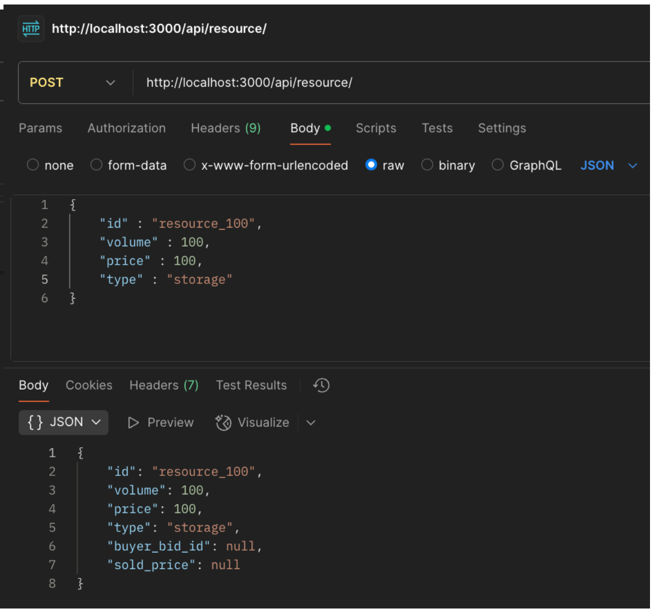
GET http://localhost:3000/api/resources/

Body: none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
```

Response ({} JSON):

```
        "sold_price": 493,
        "type": "storage",
        "volume": 96
    },
    {
        "buyer_bid_id": "bid_2",
        "id": "resource_94",
        "price": 475,
        "sold_price": 352,
        "type": "storage",
        "volume": 19
    },
    {
        "buyer_bid_id": "bid_6",
        "id": "resource_95",
        "price": 133,
        "sold_price": 491,
        "type": "storage",
        "volume": 60
    },
```



POST http://localhost:3000/api/resource/

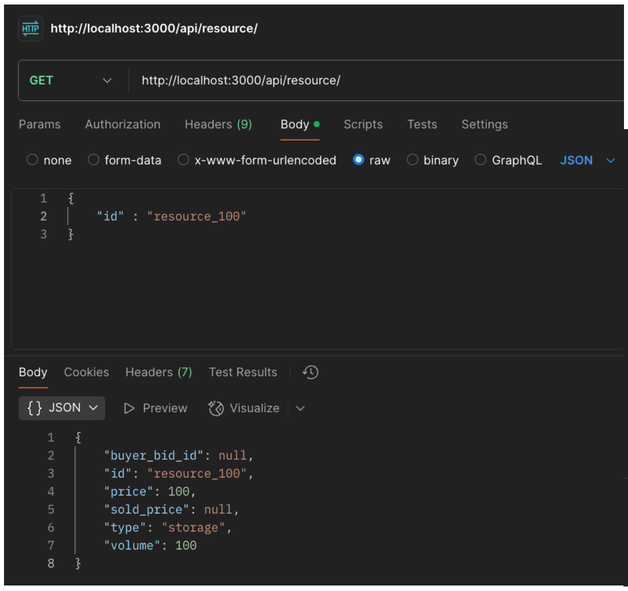
Body: none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
    "id" : "resource_100",
    "volume" : 100,
    "price" : 100,
    "type" : "storage"
}
```

Response ({} JSON):

```
{
    "id": "resource_100",
    "volume": 100,
    "price": 100,
    "type": "storage",
    "buyer_bid_id": null,
    "sold_price": null
}
```

GetAllResources() and CreateResource()



http://localhost:3000/api/resource/

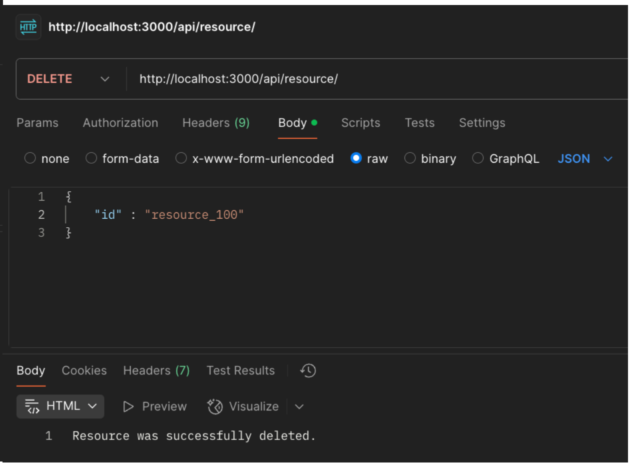
GET http://localhost:3000/api/resource/

Body: raw JSON

```
{
    "id" : "resource_100"
}
```

Response Body (JSON):

```
{
    "buyer_bid_id": null,
    "id": "resource_100",
    "price": 100,
    "sold_price": null,
    "type": "storage",
    "volume": 100
}
```



http://localhost:3000/api/resource/

DELETE http://localhost:3000/api/resource/

Body: raw JSON

```
{
    "id" : "resource_100"
}
```

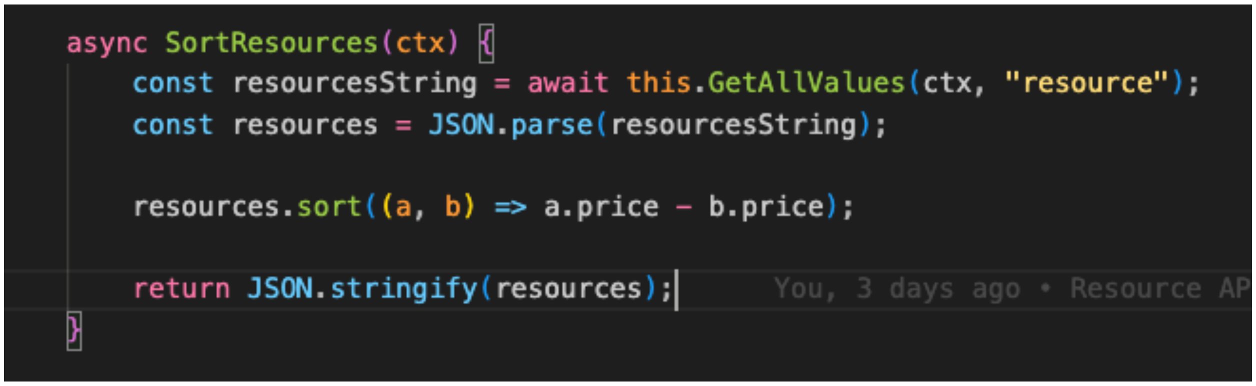
Response Body (HTML):

```
Resource was successfully deleted.
```

ReadKey() and DeleteKey()

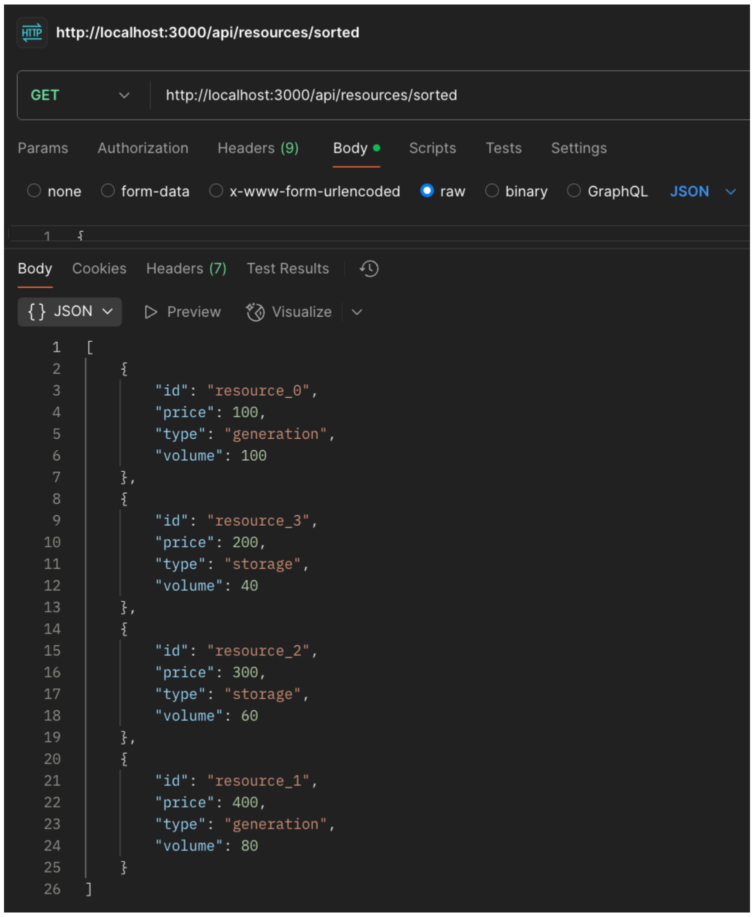
Ordering resources by price

The sortResources() function organizes all available resources based on their prices in ascending order. It retrieves the list of resources, sorts them accordingly, and returns the sorted array.



```
async SortResources(ctx) {
    const resourcesString = await this.GetAllValues(ctx, "resource");
    const resources = JSON.parse(resourcesString);

    resources.sort((a, b) => a.price - b.price);

    return JSON.stringify(resources);        You, 3 days ago • Resource AP
}
```

The result, sorted by price:



http://localhost:3000/api/resources/sorted

GET http://localhost:3000/api/resources/sorted

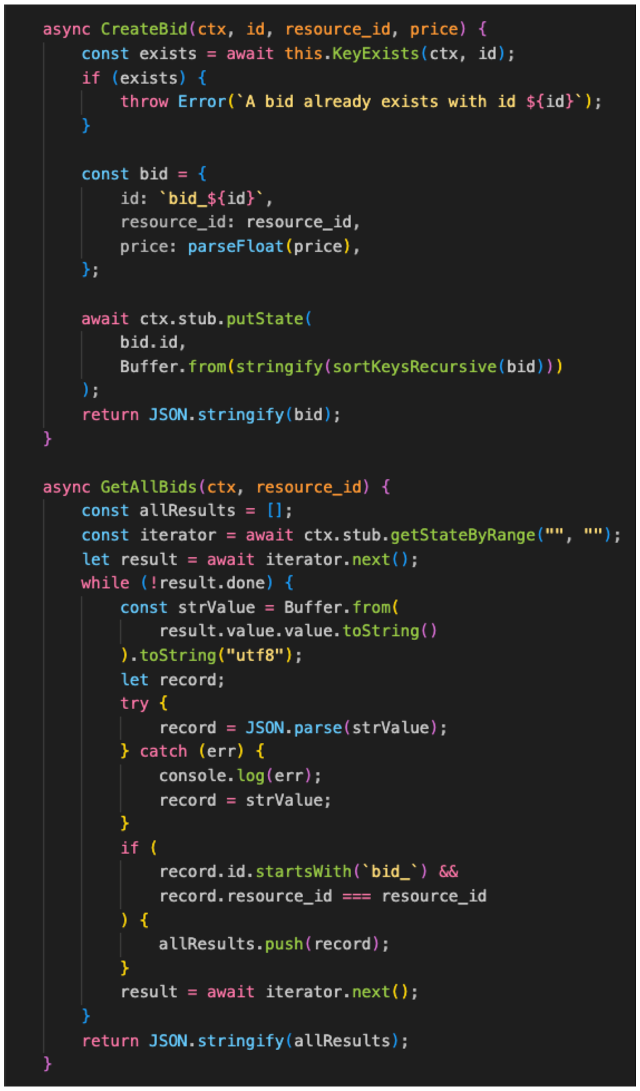
```
[
    {
        "id": "resource_0",
        "price": 100,
        "type": "generation",
        "volume": 100
    },
    {
        "id": "resource_3",
        "price": 200,
        "type": "storage",
        "volume": 40
    },
    {
        "id": "resource_2",
        "price": 300,
        "type": "storage",
        "volume": 60
    },
    {
        "id": "resource_1",
        "price": 400,
        "type": "generation",
        "volume": 80
    }
]
```

Auctions

To implement the auction mechanism, we introduce the concept of bids, which represent price offers for energy resources in the auction process. Bidders submit their bids to the network, and after a predefined period, the auction concludes by invoking either EndEnglishAuction() or EndSecondPriceAuction().

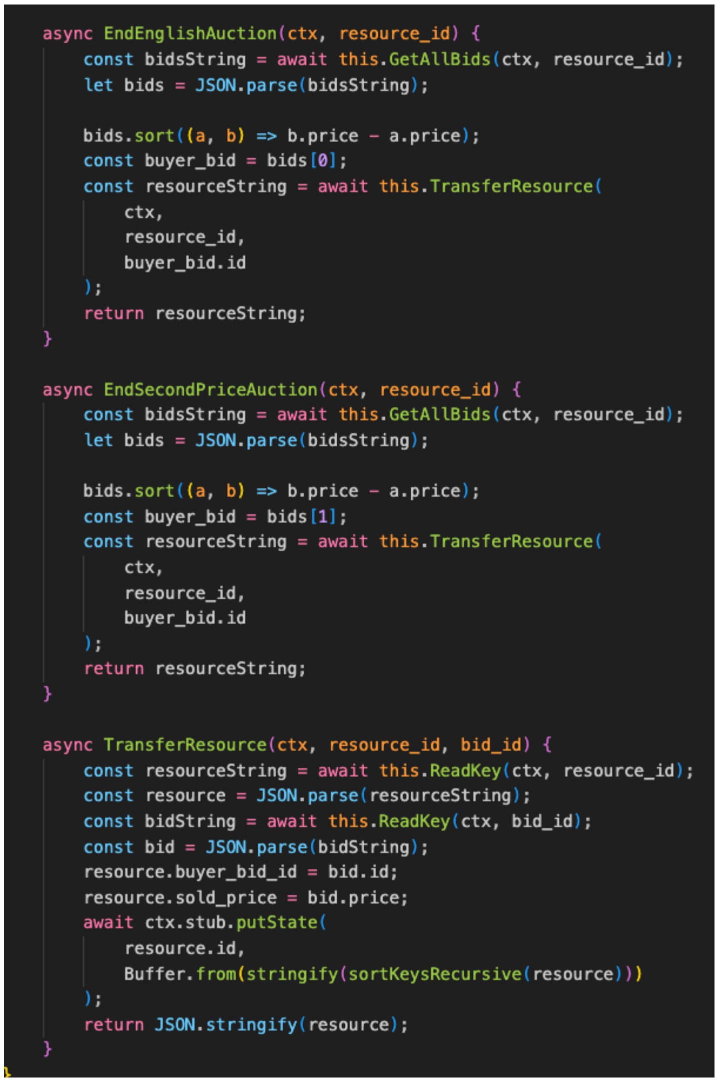
Each smart contract sorts the bids by price, and the winner is determined based on the chosen auction mechanism. Once a winner is selected, ownership of the resource is transferred to the highest bidder.

Below is the implementation of this logic, starting with the CRUD functions for bid management:



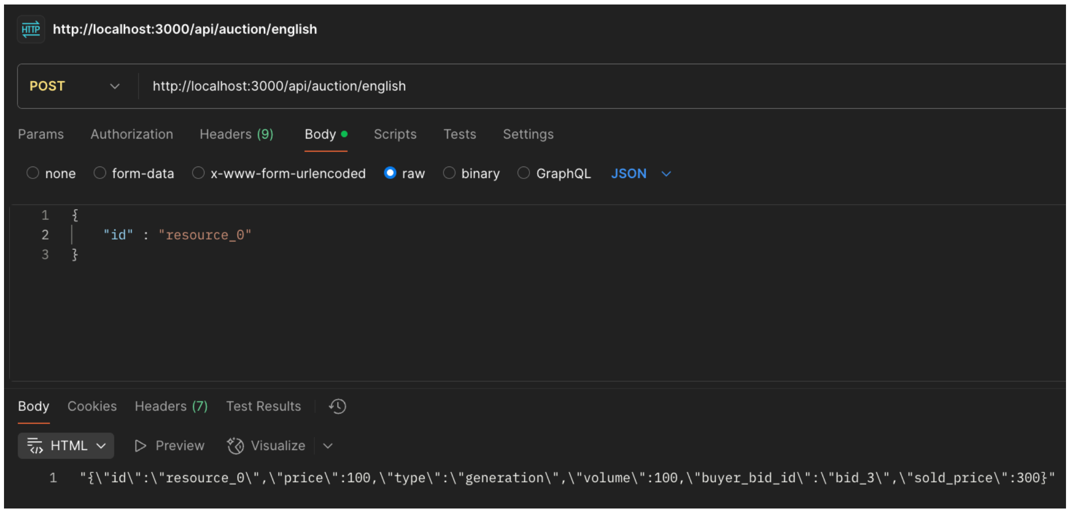
```
async CreateBid(ctx, id, resource_id, price) {
    const exists = await this.KeyExists(ctx, id);
    if (exists) {
        throw Error(`A bid already exists with id ${id}`);
    }

    const bid = {
        id: `bid_${id}`,
        resource_id: resource_id,
        price: parseFloat(price),
    };

    await ctx.stub.putState(
        bid.id,
        Buffer.from(stringify(sortKeysRecursive(bid)))
    );
    return JSON.stringify(bid);
}

async GetAllBids(ctx, resource_id) {
    const allResults = [];
    const iterator = await ctx.stub.getStateByRange("", "");
    let result = await iterator.next();
    while (!result.done) {
        const strValue = Buffer.from(
            result.value.value.toString()
        ).toString("utf8");
        let record;
        try {
            record = JSON.parse(strValue);
        } catch (err) {
            console.log(err);
            record = strValue;
        }
        if (
            record.id.startsWith(`bid_`) &&
            record.resource_id === resource_id
        ) {
            allResults.push(record);
        }
        result = await iterator.next();
    }
    return JSON.stringify(allResults);
}
```

Then, the smart contracts for auctions:



```javascript
async EndEnglishAuction(ctx, resource_id) {
    const bidsString = await this.GetAllBids(ctx, resource_id);
    let bids = JSON.parse(bidsString);

    bids.sort((a, b) => b.price - a.price);
    const buyer_bid = bids[0];
    const resourceString = await this.TransferResource(
        ctx,
        resource_id,
        buyer_bid.id
    );
    return resourceString;
}

async EndSecondPriceAuction(ctx, resource_id) {
    const bidsString = await this.GetAllBids(ctx, resource_id);
    let bids = JSON.parse(bidsString);

    bids.sort((a, b) => b.price - a.price);
    const buyer_bid = bids[1];
    const resourceString = await this.TransferResource(
        ctx,
        resource_id,
        buyer_bid.id
    );
    return resourceString;
}

async TransferResource(ctx, resource_id, bid_id) {
    const resourceString = await this.ReadKey(ctx, resource_id);
    const resource = JSON.parse(resourceString);
    const bidString = await this.ReadKey(ctx, bid_id);
    const bid = JSON.parse(bidString);
    resource.buyer_bid_id = bid.id;
    resource.sold_price = bid.price;
    await ctx.stub.putState(
        resource.id,
        Buffer.from(stringify(sortKeysRecursive(resource)))
    );
    return JSON.stringify(resource);
}
```

Results:

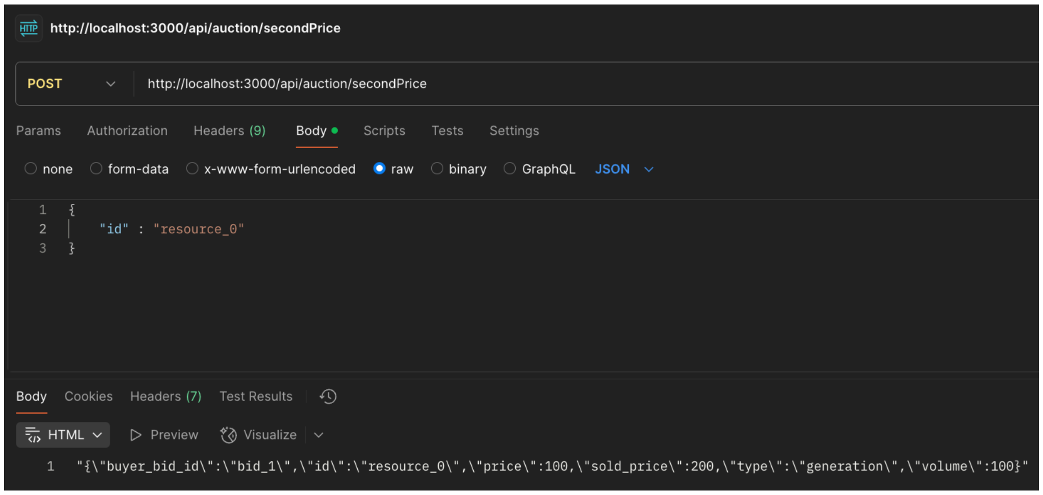


POST http://localhost:3000/api/auction/english

Body: raw JSON

```
{
  "id" : "resource_0"
}
```

```
"{\"id\":\"resource_0\",\"price\":100,\"type\":\"generation\",\"volume\":100,\"buyer_bid_id\":\"bid_3\",\"sold_price\":300}"
```



POST http://localhost:3000/api/auction/secondPrice

Body: raw JSON

```
{
  "id" : "resource_0"
}
```

```
"{\"buyer_bid_id\":\"bid_1\",\"id\":\"resource_0\",\"price\":100,\"sold_price\":200,\"type\":\"generation\",\"volume\":100}"
```

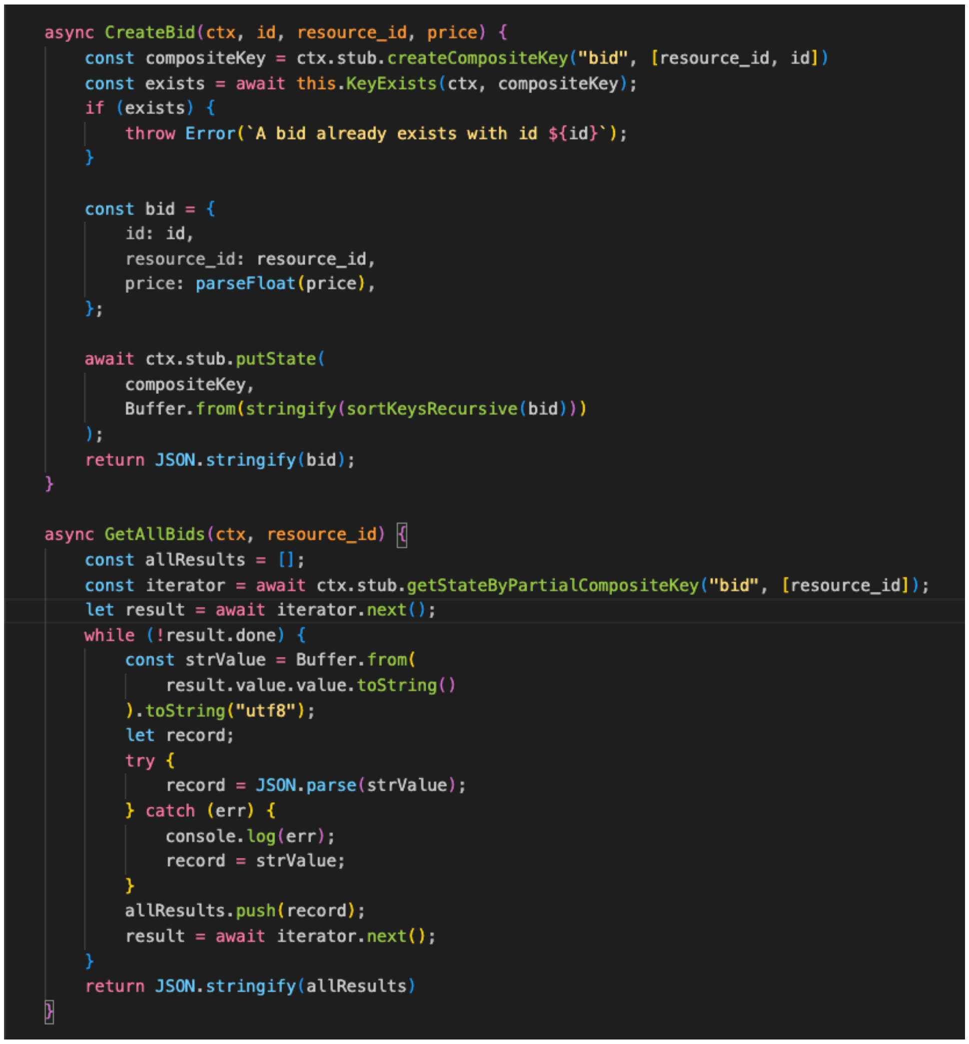
Optimization

We optimize the smart contracts using two key techniques: **composite keys** and **reducing read/write calls**.

1. **Composite Keys**: In relational databases, composite keys combine multiple attributes to enable efficient lookups. Hyperledger Fabric implements a similar concept to accelerate query performance for related attributes.

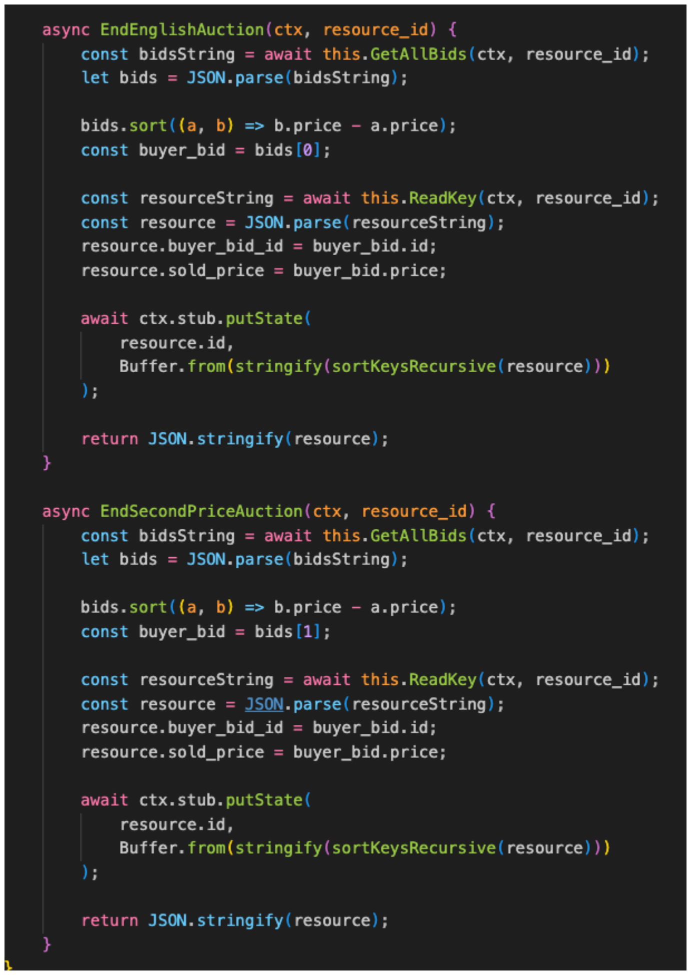
   In our implementation, each bid is associated with a specific resource, meaning the bid structure already includes the corresponding resource ID. Traditionally, retrieving all bids for a specific resource requires scanning all key values starting with "bid_" and filtering based on the resource ID. However, by utilizing composite keys—incorporating both the **bid ID** and **resource ID**—we significantly enhance query efficiency, enabling direct lookups.

Below is the optimized implementation for creating and retrieving bids using composite keys:



```
async CreateBid(ctx, id, resource_id, price) {
    const compositeKey = ctx.stub.createCompositeKey("bid", [resource_id, id])
    const exists = await this.KeyExists(ctx, compositeKey);
    if (exists) {
        throw Error(`A bid already exists with id ${id}`);
    }

    const bid = {
        id: id,
        resource_id: resource_id,
        price: parseFloat(price),
    };

    await ctx.stub.putState(
        compositeKey,
        Buffer.from(stringify(sortKeysRecursive(bid)))
    );
    return JSON.stringify(bid);
}

async GetAllBids(ctx, resource_id) {
    const allResults = [];
    const iterator = await ctx.stub.getStateByPartialCompositeKey("bid", [resource_id]);
    let result = await iterator.next();
    while (!result.done) {
        const strValue = Buffer.from(
            result.value.value.toString()
        ).toString("utf8");
        let record;
        try {
            record = JSON.parse(strValue);
        } catch (err) {
            console.log(err);
            record = strValue;
        }
        allResults.push(record);
        result = await iterator.next();
    }
    return JSON.stringify(allResults)
}
```

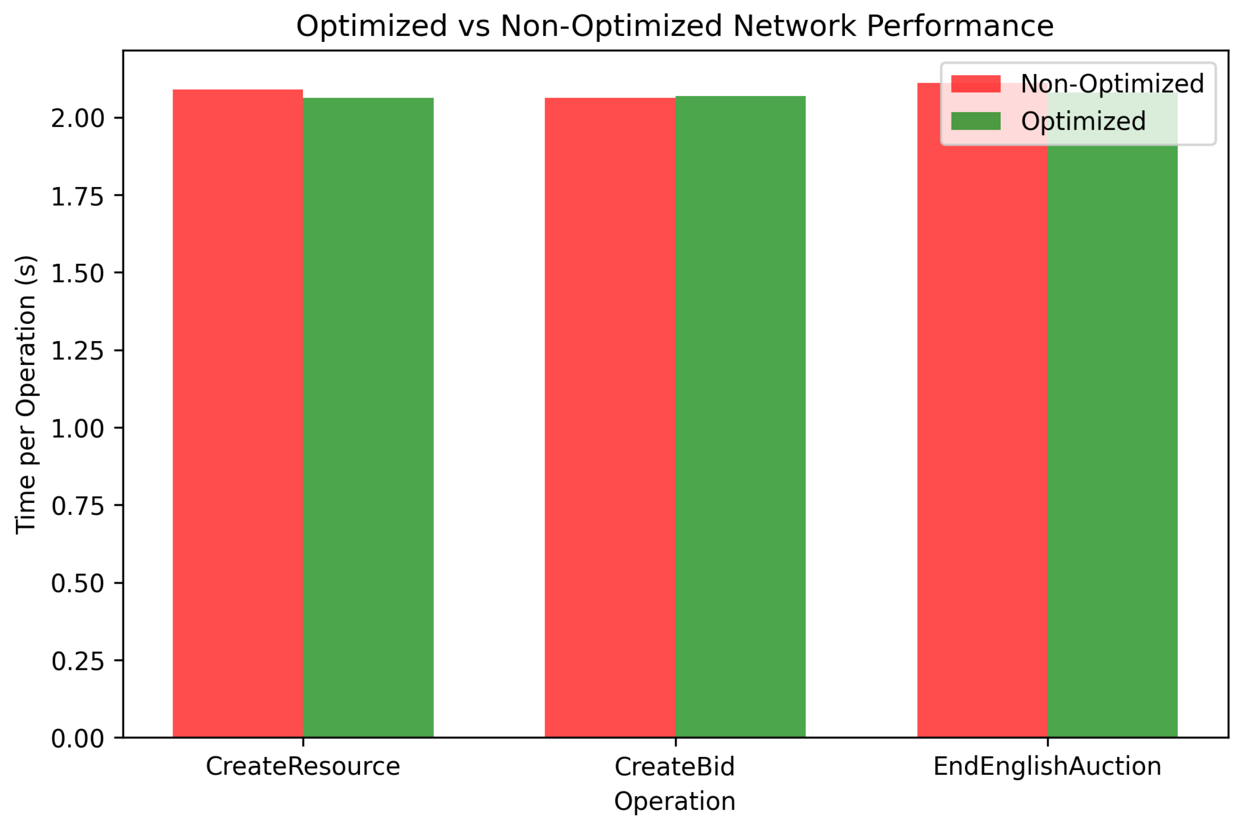
2. **Reducing Read/Write Operations:** In the current implementations of the EndEnglishAuction and EndSecondPriceAuction smart contracts, we first determine the winning bid, then pass the bid and resource ID to another smart contract to execute the transfer operation. While this approach improves the readability of our code, it introduces an extra, suboptimal read operation in the transfer smart contract. To optimize this process, we modified both auction smart contracts to eliminate the unnecessary read operation, enabling the transfer to be completed directly within the auction contract. The following is the revised, optimized code:



```
async EndEnglishAuction(ctx, resource_id) {
    const bidsString = await this.GetAllBids(ctx, resource_id);
    let bids = JSON.parse(bidsString);

    bids.sort((a, b) => b.price - a.price);
    const buyer_bid = bids[0];

    const resourceString = await this.ReadKey(ctx, resource_id);
    const resource = JSON.parse(resourceString);
    resource.buyer_bid_id = buyer_bid.id;
    resource.sold_price = buyer_bid.price;

    await ctx.stub.putState(
        resource.id,
        Buffer.from(stringify(sortKeysRecursive(resource)))
    );

    return JSON.stringify(resource);
}

async EndSecondPriceAuction(ctx, resource_id) {
    const bidsString = await this.GetAllBids(ctx, resource_id);
    let bids = JSON.parse(bidsString);

    bids.sort((a, b) => b.price - a.price);
    const buyer_bid = bids[1];

    const resourceString = await this.ReadKey(ctx, resource_id);
    const resource = JSON.parse(resourceString);
    resource.buyer_bid_id = buyer_bid.id;
    resource.sold_price = buyer_bid.price;

    await ctx.stub.putState(
        resource.id,
        Buffer.from(stringify(sortKeysRecursive(resource)))
    );

    return JSON.stringify(resource);
}
```

Comparison

To compare the two optimized and non-optimized smart contracts, we have developed a python module to run each smart contract for with 10 resources and 100 bids. Then, we call the English auction function to end the auction. We compute the average processing time of each transaction to evaluate and compare the transaction throughput of each code. The following figure shows the results of these experiments.



As demonstrated by this figure, the optimized functions are only slightly improved in terms of throughput. Nonetheless, the gap between the performance of these smart contracts will be enhanced as we scale up the network as well as the number of transactions.

	CreateResource	CreateBid	EndAuction
Non-optimized	2.1124969959259032	2.06597092628479	2.068837022781372
Optimized	2.1058998107910156	2.0644249629974367	2.067249822616577