# Sprint 2 Authentication and Authorization

Authentication, VAT, Discounts, Improved Breakdown, Cascading Threshold Calculations, and refactors

Selected Authentication Proposal

Applications communicate with the core system which provides a JWT that includes important identity information in the token payload. The TPS microservice has a simple concept of a User that references the DaAppInstall of a company.

The Microservice is capable of decrypting the JWT revealing the identity details to find the user stored in the database, so that the correct rules linked to a DaAppInstall are used for the price calculation.
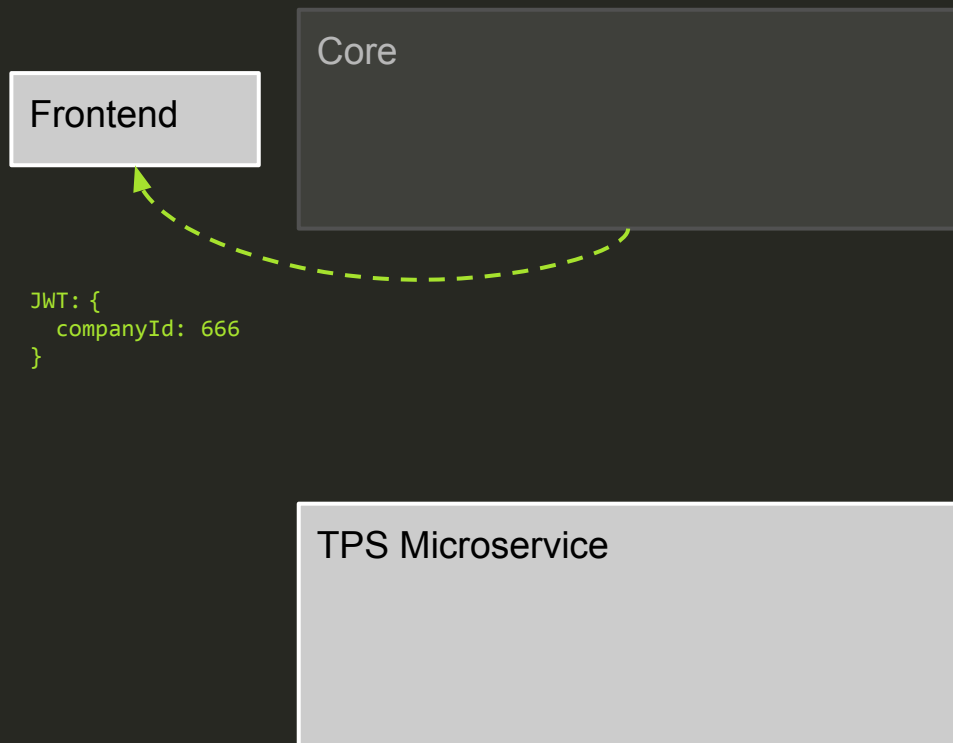

Authentication
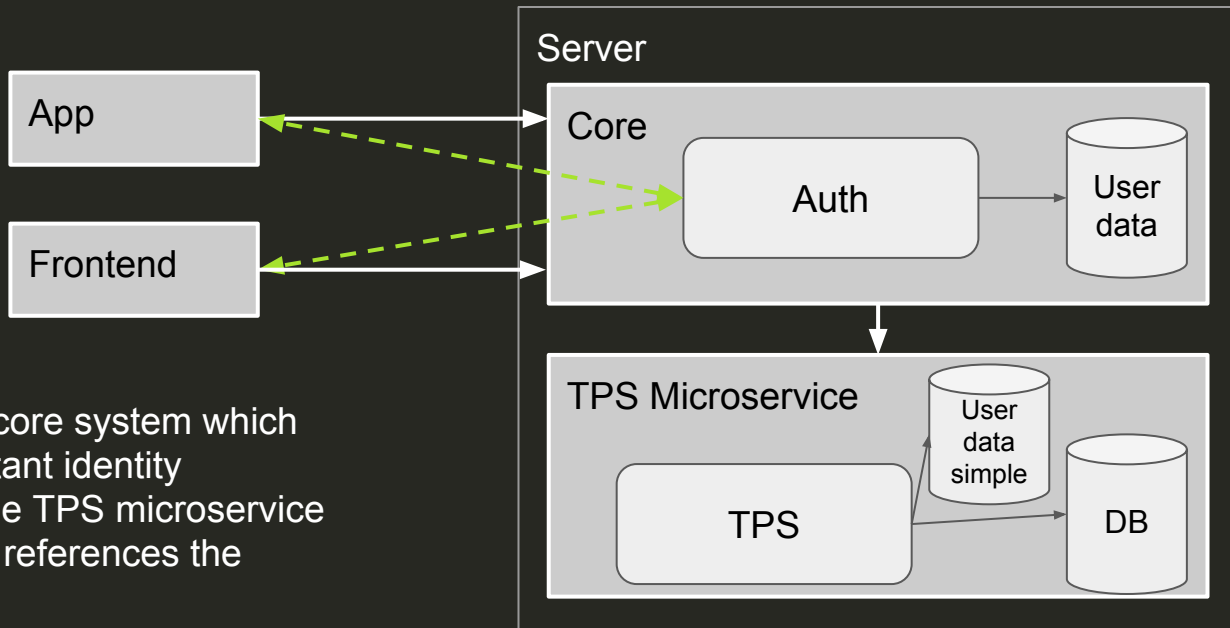
Authenticated

Authentication

Core signs JWT that
has the identity of the user
contained in the payload, e.g.:

```
const cert = process.env.JWT_SECRET;
const HOURS_24 = 86400;

jwt.sign({ companyId: '666' }, cert, {
  expiresIn: HOURS_24,
  algorithm: 'HS256'
}, (err, token) => {
  if (err) return cb(err);
  return cb(null, token);
});
```

Core

Frontend

```
JWT: {
  companyId: 666
}
```

TPS Microservice

Authentication

Frontend

Core

The frontend makes a request directed
at the Microservice, which is tunneled through the core system.
The Microservice decrypts the token to reveal the user company.

```
jwt.verify(token, cert, (err: Error, decoded: any) => {
  if (err) {
    return res.status(500)
      .send({
        auth: false,
        message: 'Invalid token provided.',
      });
  }
  console.info(decoded);
  // { companyId: 666, iat: 1521552244, exp: 1521638644 }
  next();
});
```

```
JWT:{
  companyId: 666
}
```

TPS Microservice

Authentication

Middleware:

1. Fetches token
2. Fetches secret
3. Verifies token with secret or [public key](#)
4. Adds credentials to req
5. next( )

```typescript
/**
 * Authentication middleware.
 */
export const auth = (req: Request, res: Response, next: NextFunction) => {

  // Don't use this cert in any of the warning responses, it's secret
  const cert = process.env.JWT_SECRET;
  const JWT_HEADER = process.env.JWT_HEADER || 'x-access-token';
  const token = req.headers[JWT_HEADER] : req.query.access_token;

  // It can be safe under the following circumstances:
  // 1. the JWT is one-time time usage only
  // 2. the jti and exp claims are present in the token
  // 3. the receiver properly implements replay protection jti and exp
  if (process.env.NODE_ENV === 'development' && req.query.access_token) {
    return warn(res, 449, `Don't send tokens via an URL (preferably)`);
  }

  // No token, no access
  if (!token) return warn(res, 403, 'No token provided.');

  // No secret found, still no access
  if (!cert) return warn(res, 403, 'Authentication unavailable.');

  // Verify token and add to request
  jwt.verify(token, cert, (err: Error, decoded: any) => {
    if (err) return warn(res, 403, 'Invalid token provided.');
    addCredentialsToReq(req, decoded);
    next();
  });
};
```

Authentication

1. Changes must be made to the core system
2. All communications will be held through the core system
3. Secrets must be known to both core and microservice
4. Secrets should be stored in the .env file
5. Secrets should be updated regularly

Breakdown Proposal

```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "toll": 5
    "waiting": 2.8
  }
  "currency": "EUR"
  "total": 63.58
  "tax": {
    "amount": 3.6
    "percentage": 6
  }
}
```

These values are
a sum of the
total

Breakdown Proposal

```
"price": {
  "breakdown": {
    "discount": -11.22
    "parking": 2
    "route": 65
    "toll": 5
    "waiting": 2.8
  }
  "currency": "EUR"
  "total": 63.58
  "tax": {
      "amount": 3.6
      "percentage": 6
   }
}
```

The tax is calculated but is part of the total, as VAT is included

Improved Breakdown

1. Total is the sum of the breakdown
2. Tax is included in the total, and thus included in the breakdown prices
3. Tax is based on country default tax percentage
4. The discussion is found in the breakdown proposal slides

VAT

```
// copy.country

country:
  { name: 'Netherlands',
    code: 'NL',
    defaultTax: 6,
    defaultCurrency: 'EUR' },
```

```javascript
instance.breakdown(copy)
  .then((data: Response) => {
    expect(data)
      .deep.equal({
        price: {
          breakdown: {
            route: 83,
            toll: 0,
            parking: 0,
            waiting: 0,
            discount: 0,
          },
          tax: {
            amount: 4.5,
            // 4.7 = 83 / 106 * 6
            percentage: 6,
          },
          currency: 'EUR',
          // 66.32 = 82.83 - 16.5
          total: 66.5,
        }
      });
```

VAT

1. Tax is calculated back from included VAT prices
2. Tax percentages differ per country
3. More details in the price breakdown proposal

Established formula:

```
[price / (100 + tax.percentage) * tax.percentage]
```

Discounts

```
// copy.discount
discount:
  { name: 'Discount percentage test',
    value: -20
    isEnabled: true,
    type: 'percentage',
    precedence: 88547,
    companyId: 5aa1585990e4d72312f882db }
```

```
instance.breakdown(copy)
    .then((data: Response) => {
        expect(data)
            .deep.equal({
                price: {
                    breakdown: {
                        route: 83,
                        toll: 0,
                        parking: 0,
                        waiting: 0,
                        // -16.6 = -.2 * 83
                        discount: -16.5,
                    },
                    ...
                    currency: 'EUR',
                    // 66.32 = 82.83 - 16.5
                    total: 66.5,
                }
            });
```

Discounts

1. A discount can be negative or positive
2. A discount can be disabled or enabled
3. A discount can be a fixed amount or percentage
4. A discount is calculated and is part of the breakdown total
5. Discounts are not constrained by location or timeframes yet

Cascading Threshold Calculations

```javascript
// Calculate total price
const result = Price.total(
  {
    distance: 45,
    duration: 25,
  },
  [
    {
      type: 'duration',
      threshold: 19.5,  // price after 20 min is 0.45
      value: 0.15,
    },
    {
      type: 'duration',
      threshold: 15,    // price after 15 min is 0.25
      value: 0.25,
    },
    {
      type: 'duration',
      threshold: 12,    // price after 10 min is 0.15
      value: 0.45,
    },
    {
      type: 'distance',
      threshold: 20,    // price after 20 km is 0.35
      value: 0.75,
    },
  ...
```

```javascript
...
    {
      type: 'distance',
      threshold: 15,    // price after 15 km is 0.25
      value: 0.90,
    },
    {
      type: 'distance',
      threshold: 10,    // price after 10 km is 0.10
      value: 1.25,
    },
  ],
  {
    distance: calculation(1.5),
    // Distance is 45 km
    // The first threshold is found at 10 km
    // So the first 10 km cost 1.5 per km
    // The next 5 up until 15 km cost 1.25 per km
    // The next 5 up until 20 km cost 0.90 per km
    // The final 25 km cost 0.75 per km
    //
    // Distance price is 44.5
  ...
```

```javascript
...
    duration: calculation(0.4),
    // Duration is 25 min
    // The first threshold is found at 10 min
    // So the first 12 min cost 0.4 per min
    // The next 3 up until 15 min cost 0.45 per min
    // The next 4.5 up until 19.5 min cost 0.25 per min
    // The final 5.5 min cost 0.15 per min
    //
    // Duration price is 8.1
  },
  true,
);
// Therefore result is 52.6 = 44.5 + 8.1
expect(result).to.equal(52.6);



// A function to generate a calc func with default price
const calculation = (defaultPrice: number) =>
  (price: number = defaultPrice, metric: number) =>
    price * metric;
```

Cascading Threshold Calculations

1. Either no thresholds have been provided, the normal calculation function is called straight away.
2. If thresholds are provided, there are two options:
   a. Cascade option is true
   b. Cascade option is false
3. If cascade is false, the price of the last threshold that has been surpassed will be used to calculate the price per metric.
4. If cascade is true, the first couple of km say, will be calculated with the normal price. The next km's will be calculated using the first surpassed threshold, the next km's with the next threshold price …
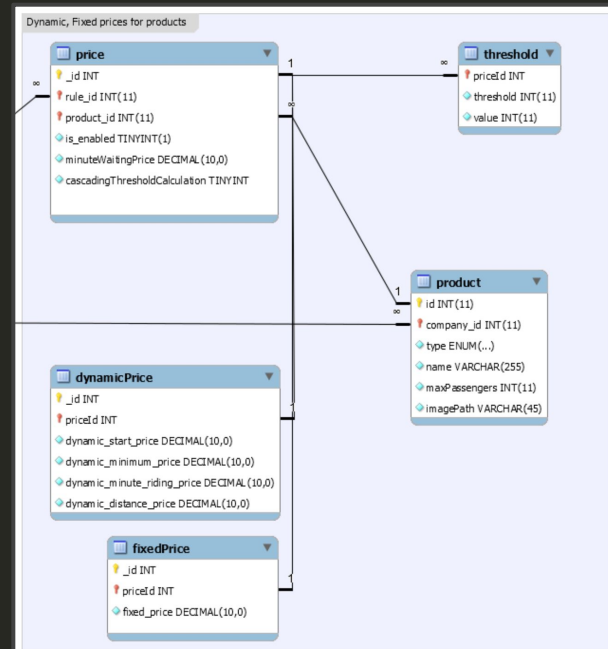
Refactors: DB Schema
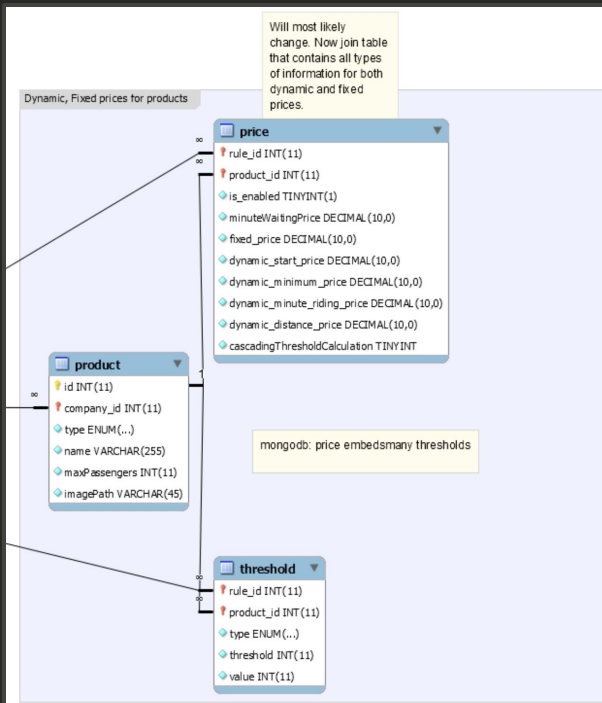
Old:
Contained pricing information of all types in one table.

New:
Separated information into individual tables while retaining core information in original price table.

Refactors: Aggregate

```ts
// util.ts
const exhaustList = (array, func, next) => {
  if (array.length < 1) return next(array);
  const pop = array.pop();
  func(pop, array, next);
};

// common/price.js
const queryPoppedVehicle = (pop, array, next) => {
  aggregateQuery(pop, (result) => {
    exhaustList(array, queryPoppedVehicle, (newArray) => {
      if (result[0]) newArray.push(result[0]);
      return next(newArray);
    })
  })
}
...
exhaustList(vehicleTypes, queryPoppedVehicle, (rules) => {
  const promises = rules.map(rule => {
    return priceCalculator.breakdown(rule);
  });
  Promise.all(promises)
    .then(breakdowns => {
      cb(null, breakdowns);
    })
    .catch(error => {
      cb(error);
    })
})
```

Old:

Executed query for every vehicle type in the vehicleTypes array recursively, limit 1.

New:

Executes one aggregate.

```js
// common/price.js
aggregateQuery(vehicleTypes, (rules) => {
  const promises = rules.map(rule => {
    return priceCalculator.breakdown(rule);
  });
  Promise.all(promises)
    .then(breakdowns => {
      cb(null, breakdowns);
    })
    .catch(error => {
      cb(error);
    })
});
```

Refactors

1. DB Schema
   a. Split the price table into three tables
   b. `price` is the junction table
   c. `fixedPrice` is the fixed price table
   d. `dynamicPrice` is the dynamic price table
   e. adf

2. Aggregate
   a. Now performs 1 query instead of sum(vehicleTypes) queries

Flow

1. Price.calculate endpoint is called
2. JWT token payload is decrypted
3. Directions service instance created
   a. Immediately fetches distance & duration
4. Price calculator instance created
   a. Directions service is passed async
5. Aggregate query is created taking vehicleTypes from body & companyId from JWT payload
6. The query is performed and the resulting pricing rules are mapped to the Price calculator instance

7. The instance calculates:
   a. totalPrice (routePrice + tollPrice + parkingPrice + waitPrice + discountPrice)
   b. priceVAT (% VAT of totalPrice)The Price calculator promises a calculation for each pricing rule (per vehicle type)
8. If a promise fails, an empty array is returned
9. Else all breakdowns are returned in an array

Flow: Breakdown Method

```
/**
 * Start price calculations. The distance and duration metrics
 * are fetched by the directionsService using an async function
 * before calculate is used to calculate the trip price.
 */
public async breakdown(pricing: pricing): Promise<breakdown> {

  Price.validPricingOrError(pricing);
  const metrics = await this.directionsService.directions();
  if (!metrics || metrics.distance < 0 || metrics.duration < 0) {
    return Promise.reject('Metrics not valid for price calculation.');
  }

  const parkingPrice = 0;
  const routePrice = Price.calculators[pricing.rules.type](pricing, metrics);
  const tollPrice = 0;
  const waitPrice = pricing.prices.minuteWaitingPrice * 0;
  const discountPrice = pricing.discount
    ? pricing.discount.type === 'percentage'
      ? percentOf(pricing.discount.value, routePrice)
      : pricing.discount.value
    : 0;

  ...
```

```
  ...
  const vatPerc = pricing.country.defaultTax;
  const totalPrice = Math.max(0, routePrice
    + tollPrice
    + parkingPrice
    + waitPrice
    + discountPrice);
  const { priceExVAT, priceVAT } = excludeVatOf(vatPerc, totalPrice);

  return Promise.resolve({
    vehicleType: <vehicleType>pricing.type,
    maxPassengers: pricing.maxPassengers,
    price: {
      breakdown: {
        route: roundHalfDecimal(routePrice),
        toll: roundHalfDecimal(tollPrice),
        parking: roundHalfDecimal(parkingPrice),
        waiting: roundHalfDecimal(waitPrice),
        discount: roundHalfDecimal(discountPrice),
      },
      currency: pricing.country.defaultCurrency,
      total: roundHalfDecimal(totalPrice),
      tax: {
        amount: roundHalfDecimal(priceVAT),
        percentage: vatPerc,
      },
    },
  });
}
```