South East Technological University

Department of Computing & Mathematics

A High-Performance Edge-Native Point of Sale Terminal with Zero Trust Security

(Architectural Integration and System Boundaries)

NAME: Vadym Melnychenko

STUDENT ID: W20108893

MODULE: ICT Placement Skills

COURSE: Higher Diploma in Science in Computer Science

April 2026

Statement of Copyright and Intellectual Property

1. Ownership and Copyright

The author and South East Technological University (formerly Waterford Institute of Technology) retain copyright of this project for academic purposes. However, the author remains the sole legal owner of the original concepts, software architecture, and source code associated with the **Wynsum**™ **RMS** ecosystem and its components, including the Wynsum POS Terminal implementation, except where explicitly referenced otherwise.

2. Proprietary Information and Trade Secrets

This document and the associated software contain proprietary information, trade secrets, and confidential algorithms that are the intellectual property of the author. The submission of this project for academic assessment does not constitute a waiver of any intellectual property rights or a license for commercial exploitation by any third party or the institution.

3. Restrictions on Use All rights reserved.

The use, reproduction, or transmission of any part of this document or the associated source code—whether electronic, mechanical, photocopying, recording, or otherwise—without the prior written consent of the author is strictly prohibited.

4. Limited Academic License

The South East Technological University is granted a non-exclusive, non-transferable license to use this material solely for the purpose of academic evaluation, moderation, and archival record-keeping. Any further distribution or application of the software components (specifically the Wynsum RMS POS Terminal logic) requires a separate licensing agreement with the author.

Project Scope clarification

I understand that my project is a part of a bigger commercial system, and in this case, I would like to clarify for the panel what exactly is in the project scope and what is not.

The Final project scope:

I started planning the Wynsum POS structure and architecture once I started the final year in the SETU as I understood that 8 weeks would not be enough time to build the POS as a final year project without a strict plan.
Wynsum POS was especially planned as a final year project.
I made it because it agrees with my business needs — I can make a final year project and at the end it will be part of my Wynsum RMS ecosystem.

So in my head, I got a simple planned architecture for the Wynsum POS. Development started at the end of December 2025 and the initial commit was made on 3 Jan 2026.

Based on this, the following components are 100% part of the Final Year Project:

- The complete React PWA architecture and source code.
- The high-performance UI logic, including the Universal Infinite List with row virtualization.
- All client-side state management (Context API/Hooks) and local persistence logic using Dexie.js/IndexedDB.
- All the context providers and business logic within the terminal interface.

Out of the Final project scope:

The Final Project Report and this document also contain information about other parts of the Wynsum RMS. Wynsum POS is not a stand-alone system and I truly believe I have to share more information about the Wynsum RMS system and the backend (Wynsum Hub) as Wynsum POS can't work independently all the time.

Based on this, the following are not presented for marking:

- The core Django backend engine and pre-existing relational database schema.
- External third-party APIs such as card payments (Stripe) or Wero.

In-scope Exception (Specific Backend Contribution)

There is one critical exception: the *ECDSA* cryptographic handshake between the POS and the backend was developed specifically as part of this final year project.
I implemented this challenge-response protocol on both the client and server sides during this course to ensure a Zero Trust security environment for the terminal. This is an additional reason why I mentioned backend logic in my reports — it was necessary to show how I secured the communication for this specific project.

Modern RMS architect explanation and why POS terminal is not stand alone

The modern market for Retail Management Systems around the world has shifted towards distributed client-server architecture, which explains why a POS terminal can no longer be viewed as a fully stand-alone project.

At the first glance, a regular RMS system and particularly POS terminals have main architecture like modern websites. It means the RMS has a POS(website) and server(backend). Additionally RMS systems usually have an Office management site - it is like an admin panel for the website.

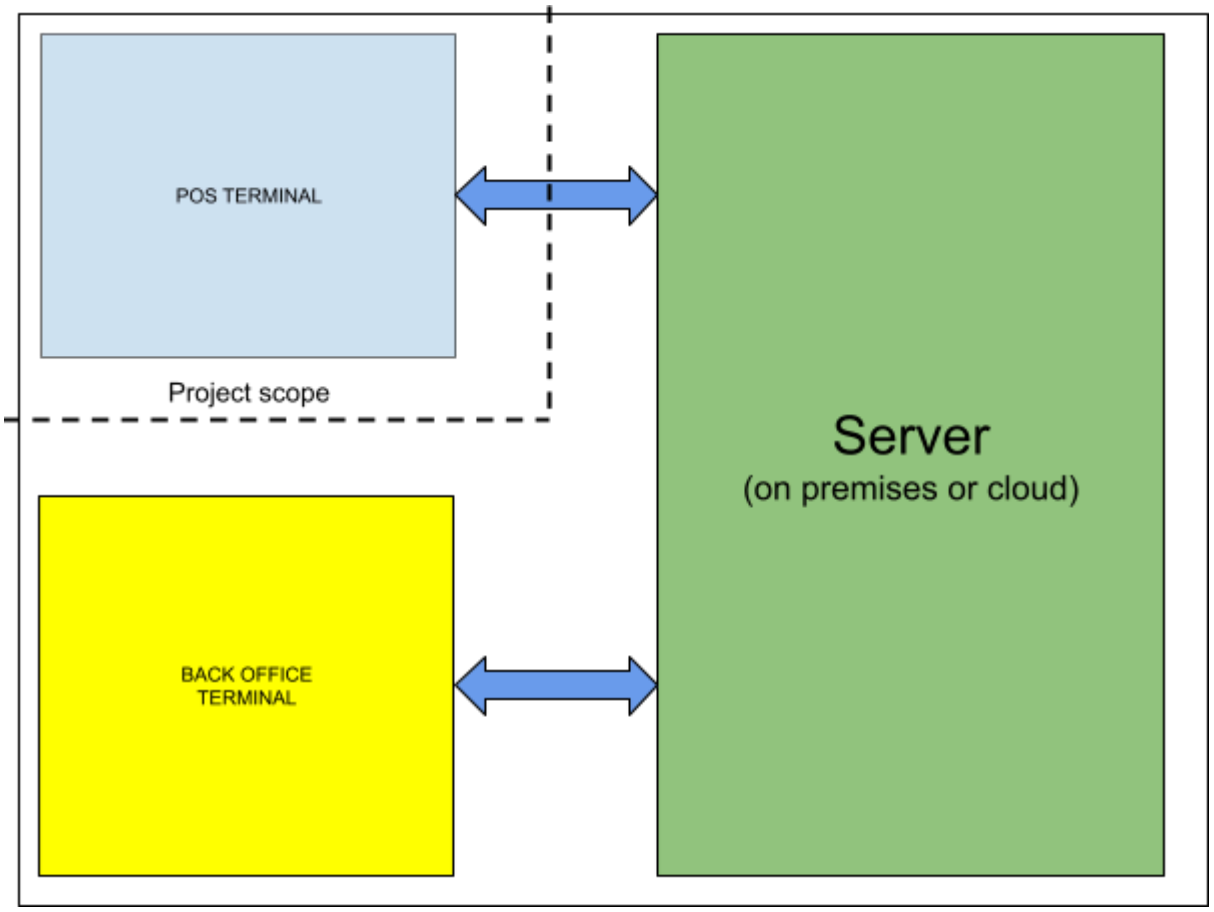

Figure 1 - The principal schema of RMS architecture

As you can see from the schema above the POS terminals are always connected to the server. And it can't work as a stand alone system. Generally it is because there could be a couple of terminals in the store and usually there are 2 and more terminals. In this case if the manager or accountant needs to count transactions and do any tax calculation, they must do independent calculation every single time, every single shift and every single day. That means it is a lot of work and expenses for business.

How modern POS terminal and RMS work

Presently on the market are two main architecture:

- on premises RMS
- cloud RMS

The first architecture - **on premises**, the POS terminal usually is a Windows app that runs on the PC.
In this case the POS terminal mostly does all the logic. For example:

- transactions
- returns
- Z reports
- X reports
- offers
- etc

Once the POS terminal finishes some of the jobs **it saves data to the database(server)**. It is a very important step - the terminal in 99.99% does not save data on the POS's PS. But many RMS like Retail Management Hero have a stand alone mode. It is needed if the terminal loses the local connection to the DB. In this case the terminal saves data locally. And of course after this the manager and accountant need to do all the calculations independently for the disconnected POS.

The second main architecture is - **cloud architecture**. In this case the terminal is "dumb", it has only a User Interface for a cashier. And all the logic on the server. The best example is - modern websites.

In this architecture if the POS terminal loses connection with the server it starts to be an expensive toy with very limited functionality. And very important - in this architecture, the server is a cloud server, so loose connection means - no Internet connection.
As you can see it is a bad choice if you live down the country or in a place where the internet cover has a bad quality.

In summary, I would like to add that the on-premises RMSs have a huge functionality and work in the local network and as result they are much more stable.

Instead of those the Cloud RMSs are beautiful and user friendly but usually have very and very poor functionality.

Why is Wynsum™ POS different?

If we start talking about the Wynsum RMS I have to mention the WRMS is fully API and cloud architectured system.

The Wynsum POS terminal, first of all is the Progressive Web App. That means it could run on all modern operating systems like Windows, Linux, Android and IOS.

The Wynsum POS is operation system agnostic.

And the main thing is that the Wynsum POS could be installed in the user operation system as a native app. As a result, Wynsum POS can work without Internet and local connection.

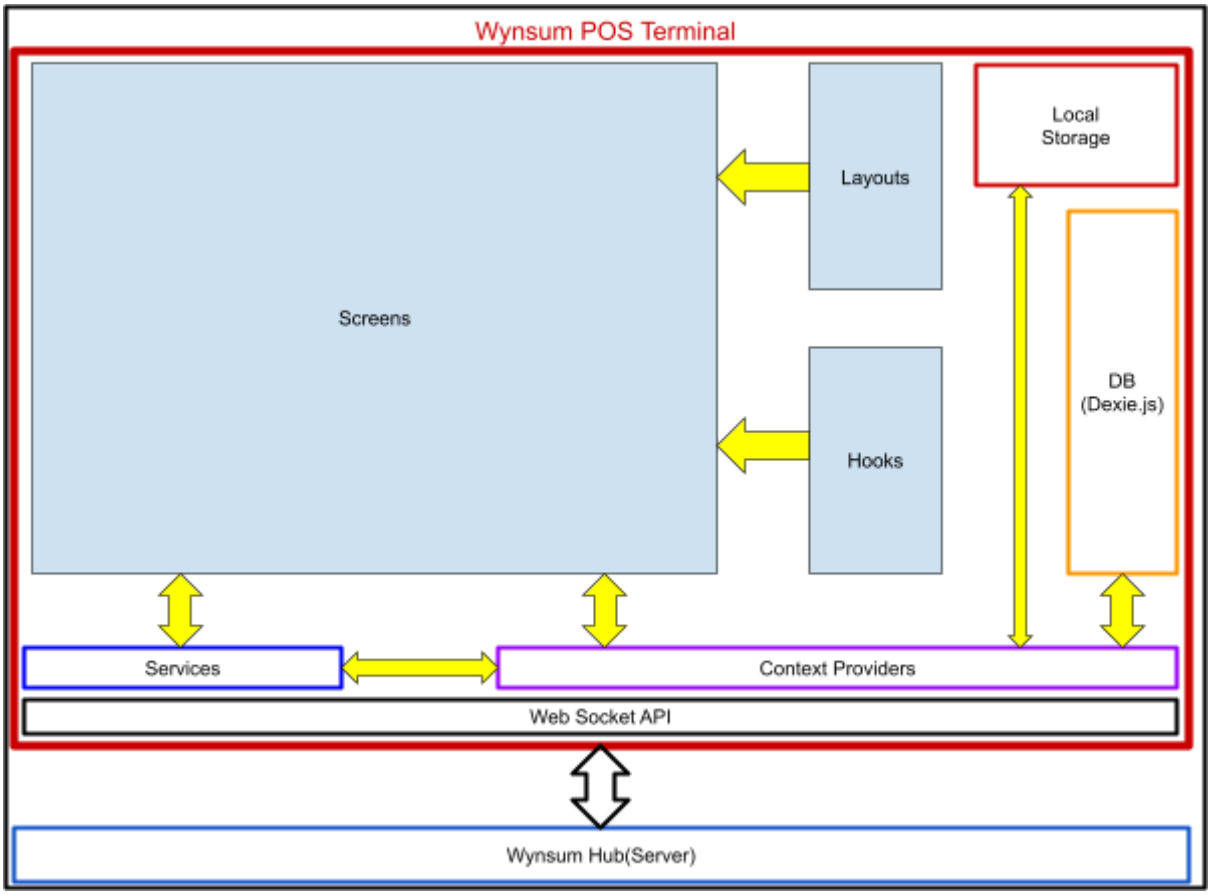

Figure 2 - Wynsum POS's architecture and Hub connection

In normal mode - with local network connection, the Wynsum POS is a "dumb" UI. All the logic sits on the server and POS provides only User Interface. But it has a couple very important features.

The first one is: **dynamic synchronisation** - every single time when the cashier scans the product, the POS updates the product's data in the local DB or saves it if the item does not exist yet.

The second one is: **background synchronisation**. Once the terminal fully connected to the Wynsum Hub, a special service takes:

- POS's settings from the server
- Cashier's settings and permissions from the server
- **Newest items updates from the server**

The Wynsum POS's architectural benefits and design boundaries

Summarising the dynamic and background synchronisations give to the POS killer benefit - possibility to work offline even if local connection is dropped.

As a result we have hybrid RMS architecture where the POS terminal is light and pretty like a cloud's systems but at the same time(if needed) it is very powerful in stand alone mode.

We also have to pay attention to the architectural principles:

- **Data Consistency Boundary:** POS terminal can't do any changes in the main(server) DB itself even if the POS terminal is online. Any changes are initiated by the terminal as a Request, but are executed and validated exclusively by the server via the **Order Pipeline**.
- **The only one source of truth is the Wynsum Hub(server)**: it controls absolutely everything. Even during the synchronisations the POS only updates data in the local DB but it doesn't change any data.
- **The payments like card payments and Wero also are not working:** because they need internet connection and must be verified on the amount of user's charge - if the store has dropped internet connection, no one can help with the online payments. And this principle follows the Wynsum architecture and previous principles.

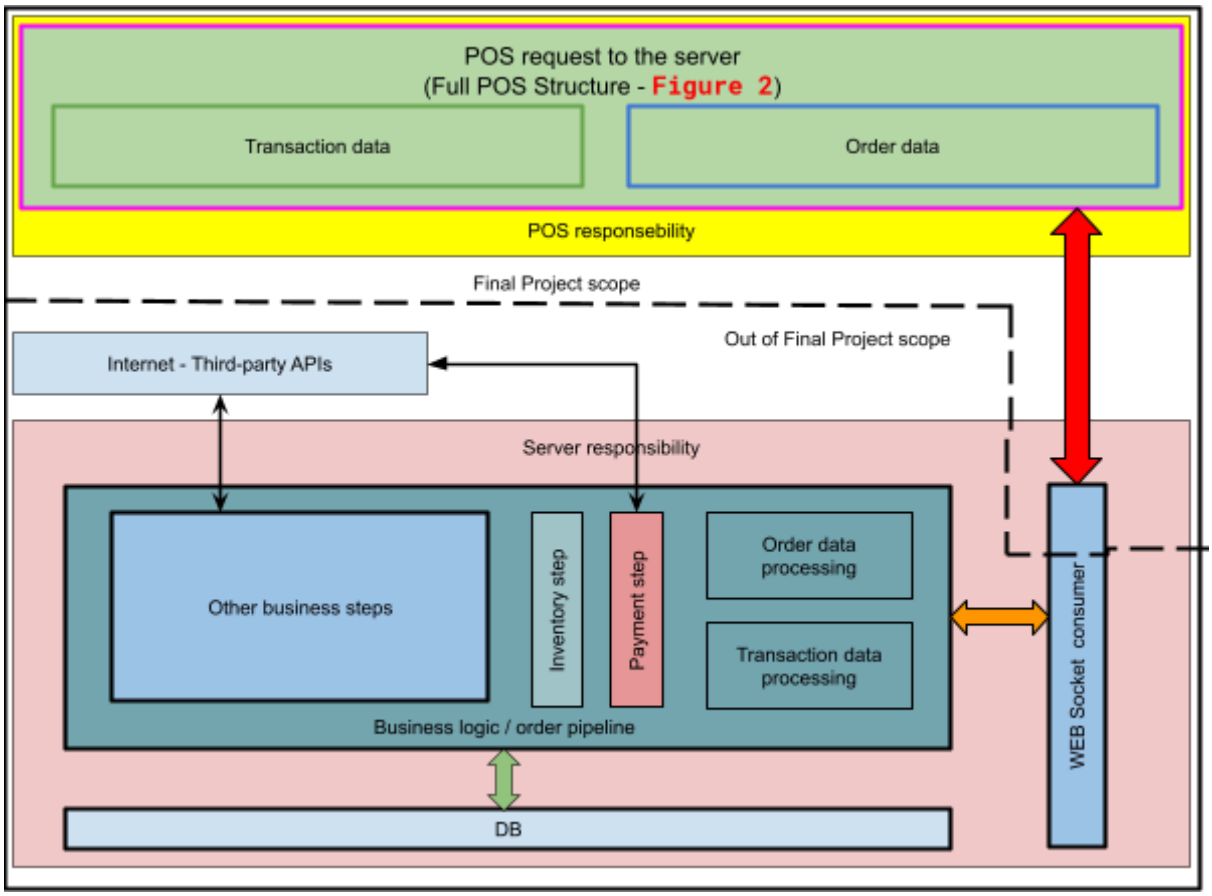

Figure 3 - POS-Server responsibilities and third party services

Data Security and network connection

The main problem in the modern RMSs is Data Security and data management. In the dinosaurs POS like RMH and RMS systems all the data sits on your local server. This is great but the data is not protected - not encrypted and passwords are not hashed(in Microsoft RMS). Even more, the connection to the SQL server is not protected and very often even has an “open access” - no password. So if you know the login - every one knows the standard SQL server login, it is - *admin*, you can connect to the local SQL server and bring happy days to the store if you drop the database.

But usually the user's data is coping and then you have phone calls from scammers. And it is a very common problem not only in retail.

And another main point - the manager or business owner has responsibility about user data and must follow **GDPR** law!

If we are talking about modern and fancy cloud systems(Square, Lightspeed) - the data protection is much better and all the responsibility about the users data is on a RSM provider. But the main problem is - **you do not own your data**.

In Wynsym the users data sits locally on the Wynsum Hub - as mentioned before it is a fully api system. So there is no SQL server someone can connect to. The Wynsum Hub has only necessary open ports. Wynsum POS gets data only with api requests to the Hub and the Hub checks if pos is authorised to get the data. Additionally only hardware authorised POS terminals can connect to the Wynsum Hub.

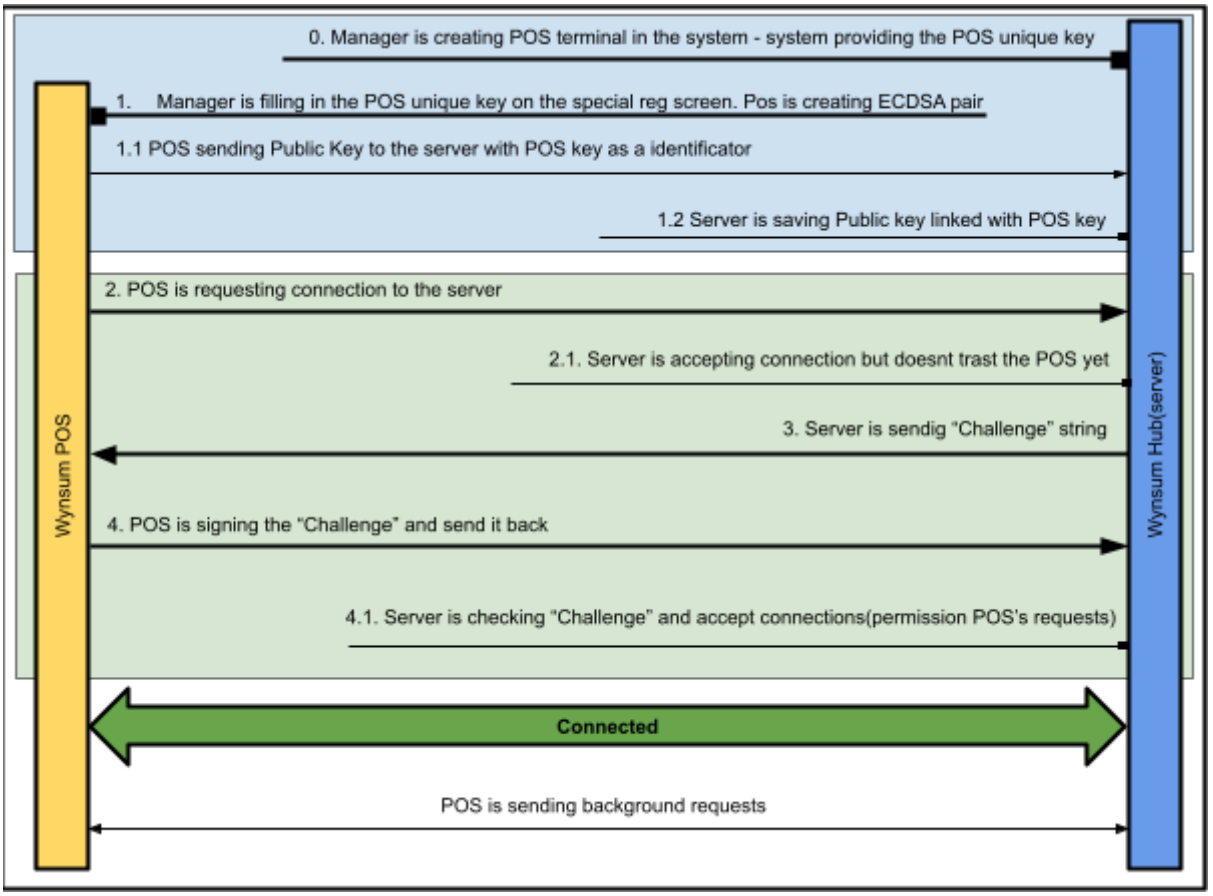

Figure 4 - Terminal “creating” and ECDSA hangshaking

Why ECDSA and why it is safe

Hardware verification - even without ECDSA the Wynsum checks each terminal with a unique key that was previously generated and then copied by manager and then filled in to the POS. It means managers manually approve each terminal.

Non-exportable keys - private keys impossible to steal using WEB vulnerabilities. It is a browser architecture.

Protection against RAM-scraping - even if a hacker knows the cashier password it is impossible to connect to the server as he doesn't have a hardware burned private key.

Backend as a SSoT - even if a hacker steals the terminal from the store, changes the POS(PWA) code and sends malicious requests - the server will not accept, as only the server does all the business logic.

Backend part of the ECDSA handshake:

```
class PosConsumer(AsyncWebsocketConsumer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.user = None
        self.device = None
        self.is_device_verified = False
        self.auth_challenge = None

    async def connect(self):
        # Always accept connection first to allow sending specific
Close Frames (e.g., 4003)
        await self.accept()

        self.device = self.scope.get("pos_device")

        if not self.device:
            print("Connection rejected: No valid device token")
            await self.close(code=4003)
            return

        if not self.device.public_key:
            # Device exists but lacks a public key for
crypto-handshake
            await
database_sync_to_async(PosSession.objects.filter(device_token=self
.device.token).delete)()
            print(f"Connection rejected: Device {self.device.name}
has no Public Key!")
            await self.close(code=4003)
            return

        await self.update_last_login()
        print(f"Device Connected: {self.device.name}")
```

```
        await self.channel_layer.group_add(
            "store_broadcast",
            self.channel_name
        )

        self.is_device_verified = False

        # Initiate Cryptographic Handshake
        self.auth_challenge = secrets.token_hex(32)
        await self.send(text_data=json.dumps({
            "type": "auth_challenge",
            "data": self.auth_challenge
        }))

    async def disconnect(self, close_code):
        await self.channel_layer.group_discard(
            "store_broadcast",
            self.channel_name
        )
        print(f"Disconnected code: {close_code}")

    async def receive(self, text_data):
        try:
            payload = json.loads(text_data)
        except json.JSONDecodeError:
            return

        # Phase 1: Handshake (Blocks all other RPC calls until
verified)
        if not self.is_device_verified:
            if payload.get("type") == "auth_signature":
                signature = payload.get("signature")
                await self.verify_device_response(signature)
            return

        # Phase 2: RPC Calls
        request_id = payload.get("id")

        try:
            method_name = payload.get("method")
            params = payload.get("params", {})

            if not method_name:
                raise RpcError(-32600, "Method not specified")

            result = await rpc_router.call(method_name, self,
```

```
**params)

            if request_id is not None:
                await self.send(text_data=json.dumps({
                    "jsonrpc": "2.0",
                    "id": request_id,
                    "result": result
                }, cls=DjangoJSONEncoder))

        except RpcError as e:
            print(f"RPC Error: {e.message}")
            if request_id is not None:
                await self.send(text_data=json.dumps({
                    "jsonrpc": "2.0",
                    "id": request_id,
                    "error": {"code": e.code, "message":
e.message}
                }))

        except Exception as e:
            print(f"Critical Error: {str(e)}")
            if request_id is not None:
                await self.send(text_data=json.dumps({
                    "jsonrpc": "2.0",
                    "id": request_id,
                    "error": {"code": -32603, "message": "Internal
error"}
                }))

    async def verify_device_response(self, signature):
        """Verifies ECDSA signature provided by the hardware
device"""
        if not signature:
            await self.close(code=4003)
            return

        is_valid = verify_signature(
            self.device.public_key,
            signature,
            self.auth_challenge
        )

        if is_valid:
            print(f"HANDSHAKE COMPLETE: {self.device.name}")
            self.is_device_verified = True
            await self.update_last_login()
            await self.send(text_data=json.dumps({"type":
```

```
"auth_success"}))
        else:
            print(f"INVALID SIGNATURE. Closing.")
            await self.close(code=4003)
```