# System #1 – Human Resources Management System (HRMS)

## Application Introduction

*Human Resources Management System (HRMS)* will be responsible for managing:

* Employees
* Salaries
* Vacations
* Payments
* Bonuses
* Promotions

## System Requirements

Functional:

* Web-based application.
* Performs CRUD operations on the employees.
* Manages salaries:
  + Allows the manager to ask for an employee’s salary change.
  + Allows the HR manager to approve or reject a request.
* Manages vacation days.
* Manages promotions & bonuses.

Non-functional:

* Classic data-driven application.

Open questions:

* *“How many concurrent users do you expect?”*
* *“How many employees are you going to manage?”*
* *“What do we know about the external payment system?”*

Answers we got from the clients:

* 10 concurrent users.
* 250 employees

Let’s start with the data volume. We are going to estimate the total size of the data the system should be able to store.

We asked our client and calculated a bit. Each employee holds around 1 MB of data. On top of that, the company stores around 10 scanned documents for every hired person – contracts, reviews, etc. One document is around five megabytes, so the total data for one employee is:

*5 MB x 10 + 1 MB = 51 MB / Employee*

The company has 250 employees currently, but we need to think long term and ask about their planned growth. Expects 500 employees in five years. The total storage required is:

*51 MB x 500 Employees = 25.5 GB*

In summary, we have the following non-functional requirements:

* 10 concurrent users.
* Manages 500 employees.
* 25.5 GB data volume forecast.

o Relational & unstructured.

* Not a mission-critical system.

## Identifying Modules

Based on our requirements, we have these initial entities – *Employee*, *Vacation*, *Salary*. The latter will include bonuses and promotions.

Based on our entities, we would define the following modules to serve our tasks:

* *Employees Service* – manages the full CRUD operations on the Employee entity.
* *Salary Service* – manages the salary approval workflow.
* *Vacation Service* – manages the employee’s vacations.

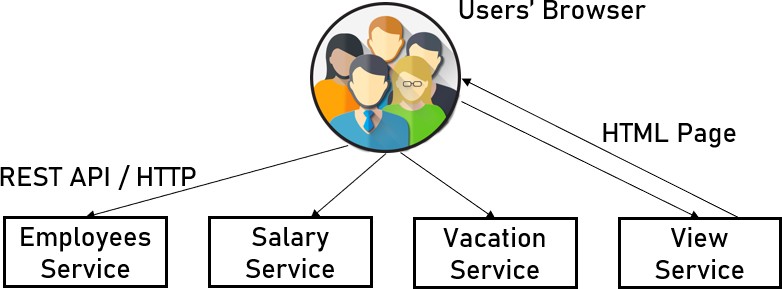
Additionally, since our application is web-based, we would need a *View Service* to return static files to the browser (*HTML*, *CSS*, and *JavaScript*).

Here is our architecture diagram so far:

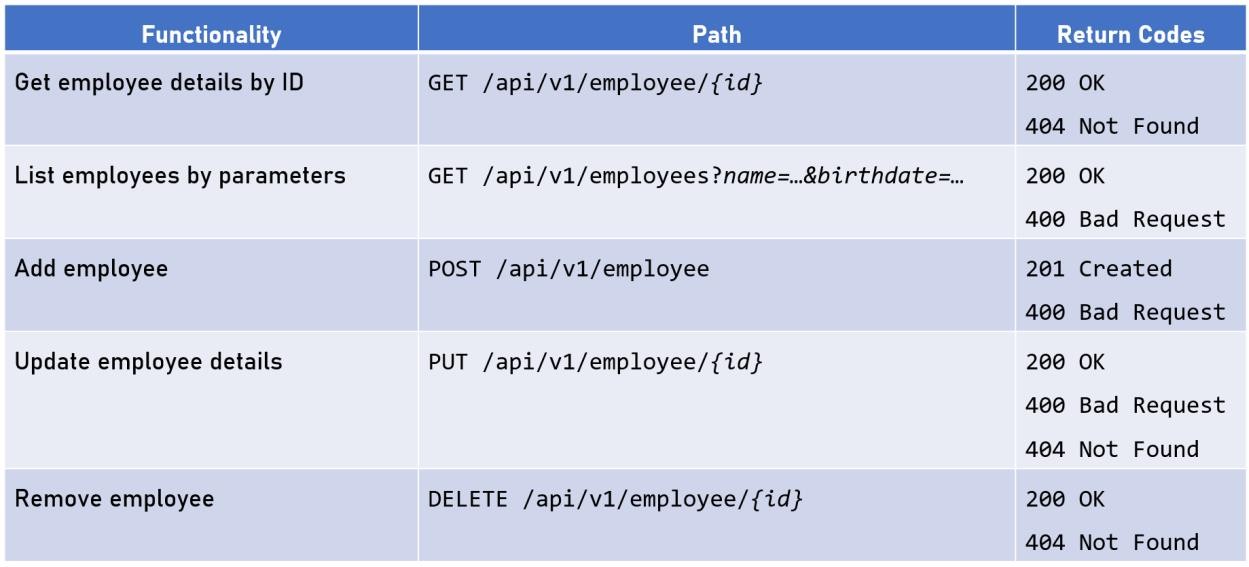


The standards for the above scenario is *HTTP* and *REST API*

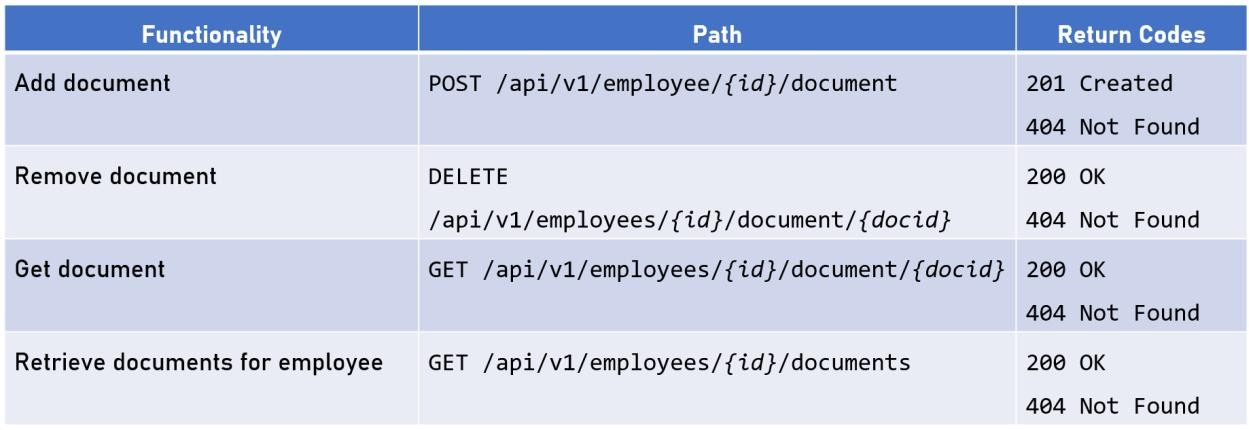
Our diagram so far:



*Employees API*:



And for the *Documents API*:



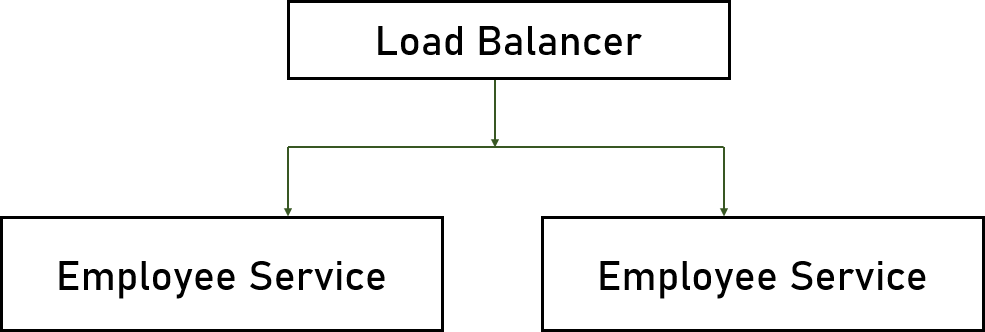
Our next consideration is about redundancy.

Pause the book and think about the redundancy here, considering what you already know.

How will you provide redundancy to the system?

I am sure you answered the question quite fast. You may continue reading now.

Since we have a stateless application, the redundancy is relatively straightforward again:



Before moving on to the next module, we should consider some cross- cutting concerns – do we need a cache, and what is our exception handling strategy?

We do not expect many queries based on our concurrent users, so no caching mechanisms are required.

As for the exception handling strategy, we are going to use the best practices for a 3-layered *REST API*:

* *Service Interface Layer* – catch, log the exception, attempt to retry, and retrow.
* *Business Logic Layer* – catch, rollback transactions, wrap additional business logic details, log the exception, and throw.
* *Data Access Layer* – catch, log the exception, log the query, and rethrow.

Of course, we will send all exceptions to our *Logging Service*.

It may seem that we are doing too much logging, but in case of a critical failure – the more debugging information we have, the easier we will diagnose the problem.

Our first *REST API* module is ready. Next - the *Salary Service*!

## Designing The Salary Service

We already covered three services in our architecture. It should be getting more comfortable for you now. The *Salary Service* has nothing special. Here are its responsibilities:

* Allows managers to ask for an employee’s salary change.
* Allows HR representatives to approve or reject the request. Pause the book. No, actually, stop it altogether!

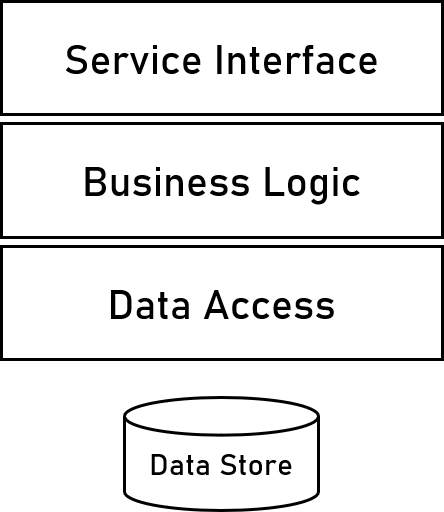
Try to design the *Salary Service* on your own. Think about what we learned so far and how we approached the architecture of the previous modules.

After you are ready, you can continue reading to find the author’s

solution.

As you may have already guessed – our application type will be a web server, and the technology stack remains the *.NET* platform with *SQL Server*.

The service architecture is also the same:



Let’s design the *REST API*. Here are the required functionalities:

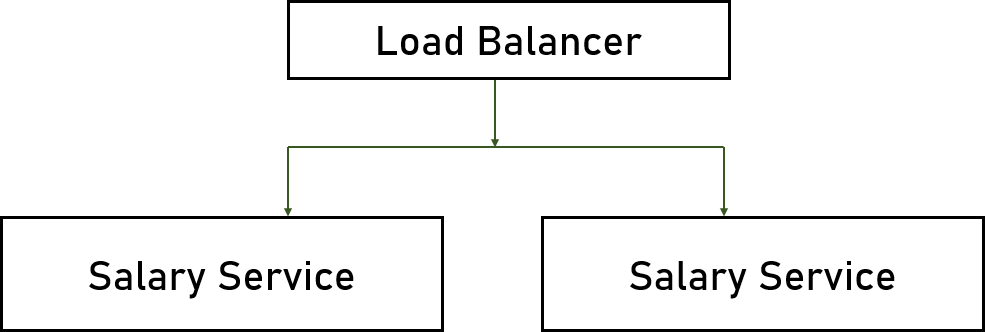
* Add salary request.
* Remove salary request.
* Get salary requests.
* Approve salary request.
* Reject salary request.

Here is the specific breakdown:



Note the “approval” and “rejection” words in the *API* paths. According to the *REST API* best practices and standards, we do not want to include verbs in the *URL*. If we used “approve” and “reject”, we would have broken the rules even though the *API* would still work, of course.

Finally, here is the redundancy solution for this service. Again, the same old load balancer concept:



As with the *Employees Service*, we do not need any caching. And we can use the same exception handling strategy as a standard throughout all our *REST API* modules in the system.

It is time for the *Vacation Service*!

## Designing The Vacation Service

This section does not contain a solution. The concepts for the *Vacation Service* are precisely the same as the ones for the *Salary Service*.

Try to design the architecture on your own. Here are the functional requirements of the module:

* Allows employees to manage their vacation days.
* Allows HR representatives to set available vacation days for employees.

After you are ready, we can continue with the *Payments Interface*!

## Designing The Payments Interface

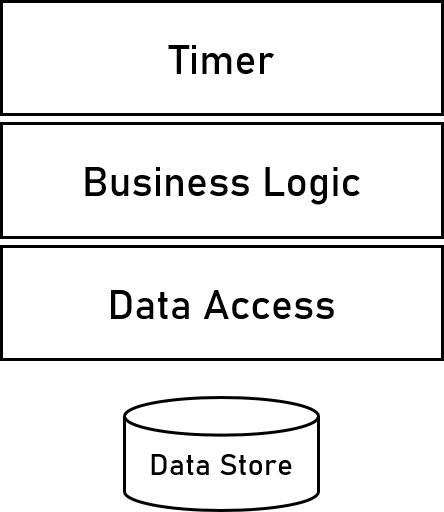
Here are the *Payments Interface* functional requirements:

* Queries the database once a month for salary data.
* Passes payment data to the external payment system. Can you guess the application type and the technology stack?

It is a service – we do not have any *UI* requirements, and the module does not expect any incoming requests. There is no reason to change the technology stack – we are staying with *.NET*.

And what about the architecture and the layers?

Instead of a *UI* or a service interface, we will need a *Timer* layer to query the database at the end of each month. All the other building blocks stay the same:



As for the redundancy attribute – we cannot use a load balancer here. We do not have requests, and the application is not a web server.

So how can we achieve redundancy for the *Payments Interface*? Think for a minute before moving on.

We will use the same *Active-Active* architectural pattern as in the *Logging Service*. It will save us any trouble with the payments. And as we know – employees love being paid on time!

It is your turn again!

What should be the exception handling strategy here? And do we need caching mechanisms?

Let’s move to the final module of our architecture – the queue!

## Choosing A Queue

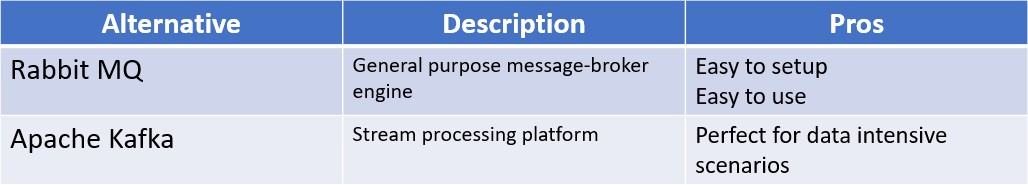
Choosing our queue is an important topic. We need to decide which one will be perfect for the logging job.

Generally speaking, we usually have three different options:

* *Self-Developed Queue*
* *RabbitMQ*
* *Apache Kafka*

Let’s analyze these options. First, we do not want to develop our own

queue. That’s asking for support and maintenance trouble. Therefore, we need to compare the other two options:



Which one would you choose?

In our system, we do not have any data streaming or intensive scenarios involved, so the main advantage of *Apache Kafka* is irrelevant for us. Additionally, *RabbitMQ* will be easier to install and use. So we have a winner!

There is one interesting question for you. Think about it.

How can you guarantee queue redundancy throughout the system? Do we need it?

The answer is quite simple – we can guarantee queue redundancy by mirroring and enabling more than one queue simultaneously. Most modern solutions like *RabbitMQ* provide such configurations out of the box, but they are very complex to install and setup. In our solution, however, the queue does not serve a crucial data role. If it is offline for some reason and cannot process messages for a couple of seconds, we will not have the full log information. The potential missing data is not mission-critical and does not justify the complexity of mirroring.

## Infrastructure Considerations

We need to consider the infrastructure and deployment configurations like:

* *CI/CD* – automated integration and delivery. *Jenkins* or

*TeamCity*, for example.

* *Containerization* – packaging and virtualization for all application parts. *Docker* is a popular technology choice.
* *Orchestration* – automatic managing, scaling and updating for all the different modules. *Kubernetes* and *Docker Swarm* are perfect for the job.
* *Environment Infrastructure* – data centers or cloud providers.

For these topics, we need to talk to our client. Most probably, they already have their infrastructure covered for their other applications.

And the client’s answer is:

*“We have our own physical servers in a data center nearby. We are running different virtual machines on them to deploy our applications. We will be using the same infrastructure for our new HR system.”*

We do not need to think about this aspect of the system. The client

has already covered their infrastructure. Now, let’s finish this project!

## Final Architecture Diagrams

Before we conclude the solution, we need to provide architectural diagrams of the system.

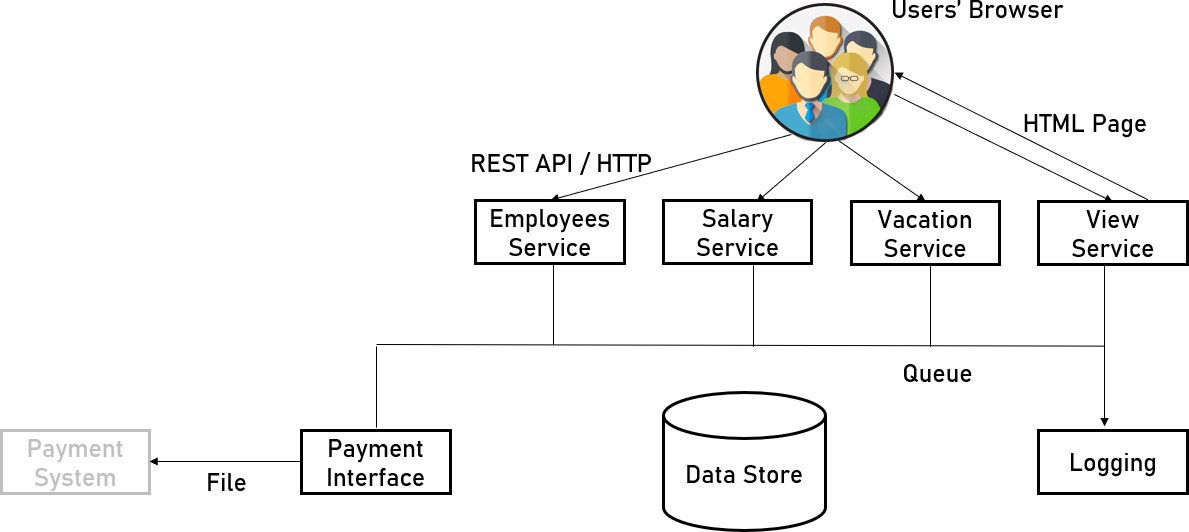
First, we need a logical diagram. It describes the different modules we designed and how they communicate with each other.

Second, we need a technical diagram. It shows what the technology stack is for each service.

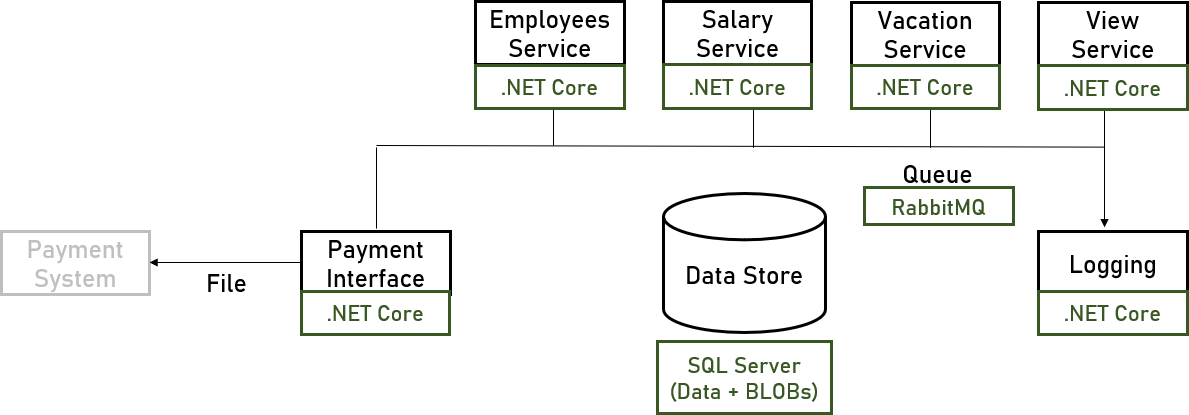
Finally, we have a physical diagram. It depicts the redundancy of each module and how to develop and deploy it on the hardware.

As we mentioned earlier, the REST API services are logically separated, but the actual physical web server may contain only a single web application, as it is in our case.

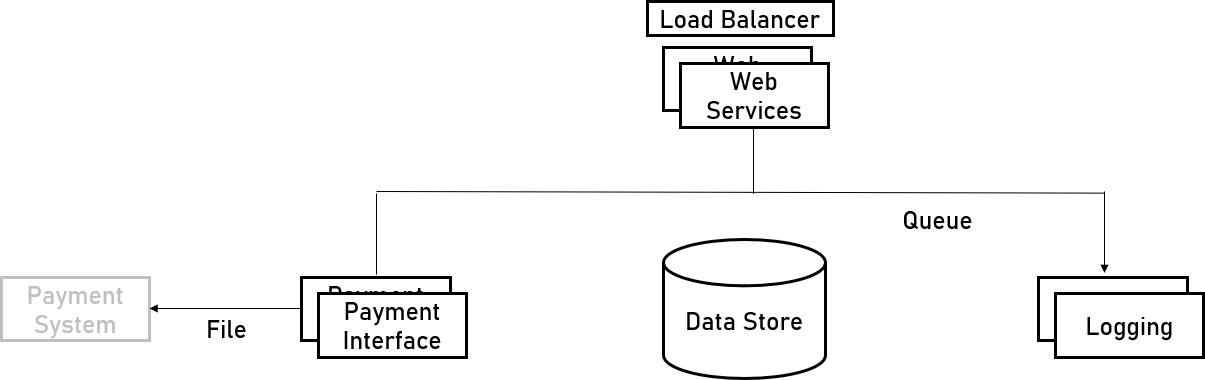
*Logical Diagram*



*Technical Diagram*



*Physical Diagram*



Our solution for *Paper & Son Limited* is now complete! Good job!

# System #2 – Bradva Cars

## Application Introduction

*Bradva Cars* is a manufacturer of autonomous vehicle systems. They can take a completely standard car and upgrade it with additional hardware and software to make it driverless on public roads. *Bradva Cars* currently have more than 10,000 vehicles on the streets but expect rapid growth to 200,000 by the end of the next year. Their CEO Melon Husk hired you to design a new system that reliably collects and visualizes live telemetry data from their installed devices.

## System Requirements

Like with the previous project, we need functional and non-functional requirements. Here is what we already know from *Bradva Cars*:

* Web-based application.
* Receives telemetry from cars (location, speed, breakdowns, and many more).
* Stores telemetry in a persistent store.
* Displays dashboards summarizing the data.
* Performs analysis of the data.

Let’s move on to the non-functional requirements and think about what we already know:

* It’s a data-intensive system.
* Not a lot of end-users.
* A vast amount of data.
* Performance is important.

Pause a the book and think about the questions you would ask Bradva Cars. What are the most important non-functional requirements for this system?

After an exciting conversation with Mr. Husk, we now have the following answers:

* *“How many concurrent users do you expect?”*
  + 10 concurrent users.
* *“How many telemetry messages do you receive every second?”*
  + 7,000 messages / second.
* *“What is the average size of a message?”*
  + The average size of a message is 1 KB.
* *“Are the messages schema-less?”*
  + Yes, the messages are schema-less.
* *“Can we tolerate some message loss?”*
  + Well, sort of…
* *“What is the desired SLA?”*
  + The highest possible.

Let’s calculate the data volume of the system. 1 message is 1 KB on average, so with 7,000 messages per second, we have:

*7,000 Messages / Second = 7 MB / Second 7 MB / Second = ~25 GB / Hour*

*~25 GB / Hour = ~605 GB / Day*

*~605 GB / Day = ~221 TB / Year.*

By any standards, 221 TB per year is quite a lot. There is no database with the current technologies that will handle such sizes without specific tuning and advanced operations.

But before we decide to bring in an extraordinary database cluster to the technology stack along with a large army of DB administrators, let’s think for a minute.

When we have vast amounts of data on a system, we always need to consider the stored information’s retention period. After all, how can such a volume of information be always relevant?

You may ask what a retention period is. The retention period defines how long we need to keep some data in our database before removing it from there. But what happens afterward? We either delete or archive the records. It depends on the business scenario.

What should motivate us to consider retention policies for our data? Two main reasons:

* Keep the database healthy and far away from exploding.
* Keep the queries’ performance fast and optimized.

The retention period solution is nothing new. Lots of cloud storage providers implement it as part of their services.

Our next step now is to talk again to Mr. Husk and ask him about the data. He responds:

*“We need the data for two operations. The first one is for our near- real-time analysis for specific cars, and the second one is for aggregated business intelligence.”*

As we learned, there are two data-driven scenarios in *Bradva Cars*. We need to discuss with them the retention periods:

* *Operational Data* – for monitoring real-time information from cars on the streets. The performance is mission-critical here. The retention period is 1 week.
* *Aggregated Data* – for business intelligence and reports. The data is not real-time, so querying it can be slower. There is no retention period here. We should keep the information forever.

Let’s return now and recalculate our volume for the operational data:

*~605 GB / Day = ~4 TB / Week*

4 TB a week is much simpler to manage, especially when we add a retention period. Our database will not have more than 4 TB of data at any given point. It is still a lot, but we can deal with it.

So far, so good. Let’s analyze the non-functional requirements of the

*Bradva Cars* system:

* 10 concurrent users.
* 7,000 messages per second.
* 4 TB maximum data in the operational database.
* Mission-critical system.
* Performance is a must.

Sounds interesting. We are now ready to identify the system’s key

scenarios and map its main modules!

## Identifying Modules

Now, pause the book for a minute and try your architecting skills once again:

* How many different modules are you going to add to your solution? Define them based on the essential requirements.
* What are the main challenges of the system, and how are you going to handle them?
* How are you going to implement the data storage and the retention policy?
* How are you going to connect the modules? How are they going to communicate with each other?
* Answer carefully to all the questions above and try to create a diagram for your architecture.

After you are ready, you can continue reading to find one of the possible solutions.

Based on the requirements, these are our key scenarios:

* Receive telemetry.
* Validate telemetry.
* Store telemetry.
* Query & analyze telemetry.

First, we need to put the *Cars* module on our architecture. Even though we do not design that specific software, it is the source of our system’s data.

The next module is the one that receives the telemetry from the cars.

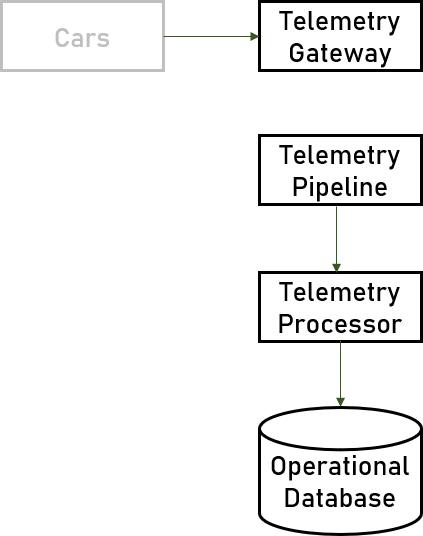
Let’s name it *Telemetry Gateway*:



It is important to note that the *Telemetry Gateway* will be only responsible for receiving the data. It will not validate it or store it in a data store. The reason is simple. The system’s load is so huge that we want this module to be as thin and small as possible. The more processing we add, the bigger the chance for crashes under the hefty request size.

Our next module is the *Telemetry Pipeline*. Its role is to queue the messages for future processing. It should do at a steady and controlled pace. This way, the system’s whole load will be only in the gateway module and nowhere else. Any other parts of the solution can poll the queued messages in a much more controlled manner.

After the pipeline, there will be our next module – the *Telemetry Processor*. It will retrieve data from the queue, validate it, process it, and finally – it will store it in our database:



Afterward, we need a *Telemetry Viewer*, which will query the database and display information in real-time. This module will visualize the data the way the end-users require it.

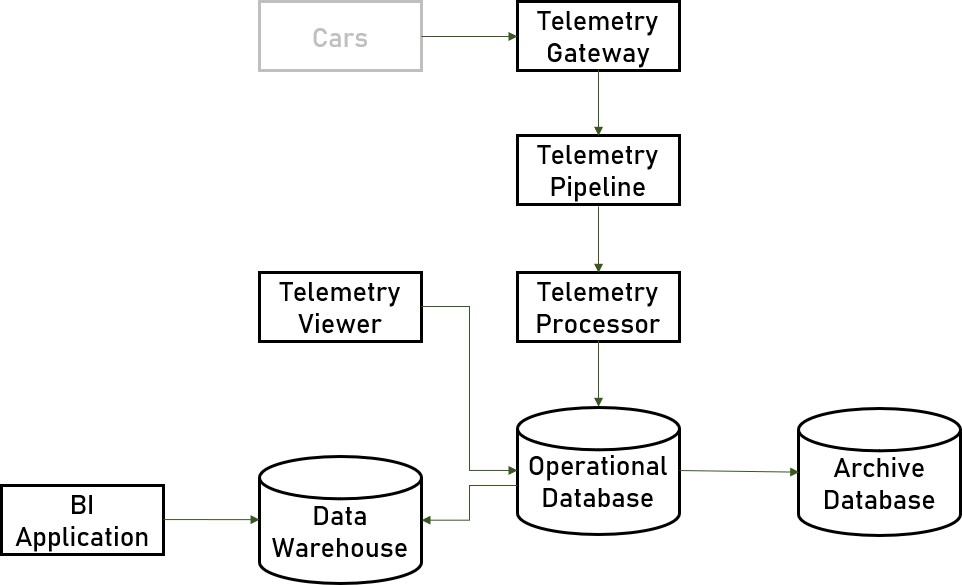
Next to the operational database, we will introduce a *Data Warehouse* database, responsible for data aggregation. It is always a good idea to store the real-time data and the reports-related one in separate databases. If we don’t do it, our business intelligence analysis may hinder the real-time dashboards’ performance.

Of course, we will also need a *BI Application* module to generate reports by querying the *Data Warehouse*.

Finally, we need to design our 1-week retention policy. We just need to introduce a third database storing all archived records. Let’s name it *Archive Database*. The only requirement for it is the crucial ability to store huge amounts of data. Working with archived data usually is so rare that we do not need to think about any performance considerations.

Great! We identified all modules in our system!

Here is the complete architecture of the *Bradva Cars* application:



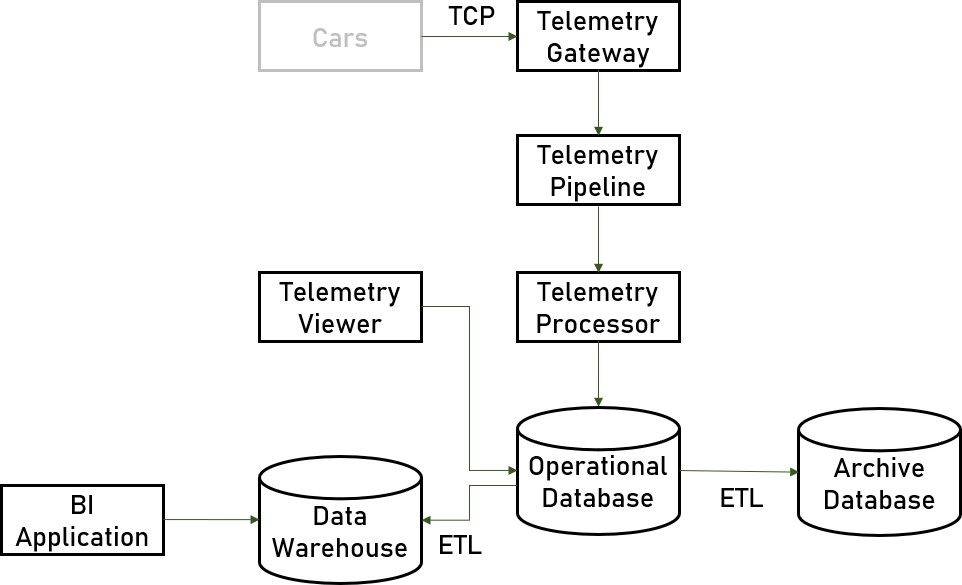
It’s time for messaging. We need to discuss how these modules will communicate with each other, and we need to decide the transport mechanisms behind each service’s contract.

The first protocol we are going to design is the one between the cars and the gateway. Most *Internet of Things* devices work with the low- level *TCP* protocol. It is fast, reliable, and perfect for small bandwidth with lots of data environments.

The second protocol is the one between the databases. We will use *ETL (Extract, Transform, Load)* – a specialized process for moving data. It extracts data from one database, transforms it if required, and loads it into another database. It is not a pure communication protocol, but it is worth considering on this architecture step.

We will stop here and not discuss the protocols between the *Telemetry* modules because the technology of the *Pipeline* itself will dictate it.

Here is the communication diagram so far:



Great. It’s time to start designing our modules. Let’s start with the

*Telemetry Gateway*!

## Designing The Telemetry Gateway

The *Telemetry Gateway* is a relatively simple module, but a critical one. Its functional requirements are:

* Receives telemetry data from cars using *TCP*.
* Pushes the telemetry data to the pipeline queue.

As usual, we start with the application type and the technology stack. Pause the book and answer the following questions:

* Which application types are the best possible options for this module?
* What will be the best technology stack here? After deciding, you can continue reading.

The application type is not a web server because, typically, web servers require *HTTP* communication. We do not need any UI or visualization, so the perfect choice here will either a console application or a service. Both options are suitable.

Before picking a technology stack, we need to analyze these considerations:

* Load of 7,000 messages per second.
* Performance is a must.
* The team’s current knowledge.
* Environment and infrastructure.

Let’s first ask *Bradva Cars* for their current technologies, and we will decide. Their answer is:

*“Our developers are familiar with Python and are experts in JavaScript. In addition, we use only Linux servers in our company.”*

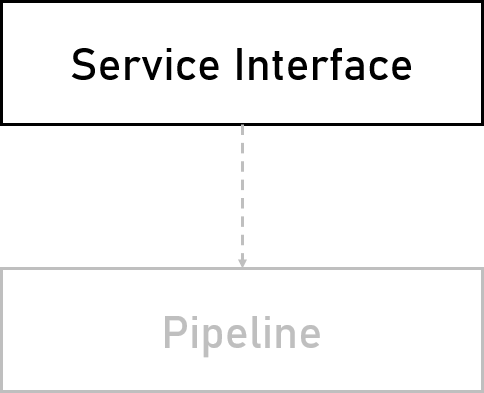
The primary server-side expertise of the company’s developers is *Python*. However, we cannot use *Python* for this module, as it is relatively slow. We are expecting a huge load and must provide outstanding performance, which *Python* simply cannot deliver. It is interpreted language, and it doesn’t have the performance optimizations available in other platforms.

We should look for another option. It must run on *Linux*, be fast, and leverages the team’s skills. It seems like the perfect technology candidate for this module is *Node.js*.

Great, let’s move to the inner architecture. The traditional 3-layer architecture is not suitable here. We do not have any data access, and there is no business logic.

All the module needs to do is get a message and push it as quickly as possible to our pipeline.

Here is a visualization of our architecture:



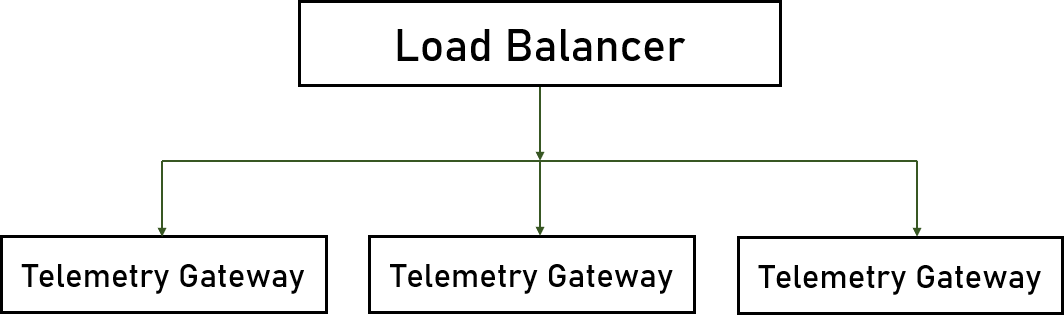
As we can see, it is quite a simple one – exactly what we need for a module dealing with such a high load.

What about redundancy? How would you handle it? Pause for a bit and test your architectural skills.

We have already seen a redundancy solution suitable for the current scenario, and it involves a load balancer in front of replicated instances of the module.

However, there is a small difference compared to the previous system we designed. Last time we used only two instances because redundancy was everything we cared about. With the vast data load here, we also need to handle all requests promptly.

But how many instances do we need? The rule of thumb for this question is to start with at least 3 and then scale out as needed:



What about other cross-cutting concerns? Logging, exception handling, caching? How would you approach them in this critical module?

After you answer these questions, we will be ready to move on to the next module – the *Telemetry Pipeline*!

## Designing The Telemetry Pipeline

Here are the functional requirements of this module:

* Gets the telemetry messages from the gateway.
* Queues them for further processing.

Basically, we need a queue for streaming high volume data. Can you think of one?

The answer lies in the previous’ system considerations.

The word “queue” here hints that we might be able to use a third- party mechanism.

But we cannot be sure. Firsts things firsts, two questions are lying in front of us:

* Is there an existing queue mechanism in the company?
* If no, should we developer our own queue?

Asking the customer leaves us quite a freedom because *Bradva Cars* does not use any queue technology for their processes. As for the second question, the answer will always be “no” because the third- party solutions do a fantastic job.

The best candidate for high load queue processing is *Apache Kafka*.

Let’s consider its pros first:

* Top-rated engine.
* Can handle a massive amount of data.
* High availability support. And the cons:
* Complex setup.
* Complex configuration.

Should we use it? What do you think?

Of course. After the initial complexity, it will be the best tool available for our requirements. The *Telemetry Pipeline* component is quite essential for our system to make any sacrifices. Additionally, *Apache Kafka* comes with built-in availability support.

The first two modules are now designed successfully. Next, the

*Telemetry Processor* and its two databases!

## Designing The Telemetry Processor

These are the functional requirement of the *Telemetry Processor*:

* Receives messages from the pipeline.
* Validates and processes them.
* Stores them in a data store. Your turn now. Pause and think!

What is the application type of this module? What would be the perfect technology stack?

You should be getting familiar with the architectural process and the general concepts by now.

We will be using a console application or a service again. This module’s communication is with the *Apache Kafka* queue, which requires a long-running task and polling.

Now, we need technologies for the processor itself, the operational database, and the archive.

Let’s start with the processor. The choice is simple enough. We can use *Node.js* again. It is fast, it has excellent *Apache Kafka* support, and is already part of the system.

Next, our operational database. We need schema-less messages because our data is unstructured by design. We also need quick retrieval and fast performance. Additionally, it looks like we will not have any complex queries. We should visualize real-time data – there should not be any advanced joins or filterings involved. All of these requirements point to a very popular database – *MongoDB*.

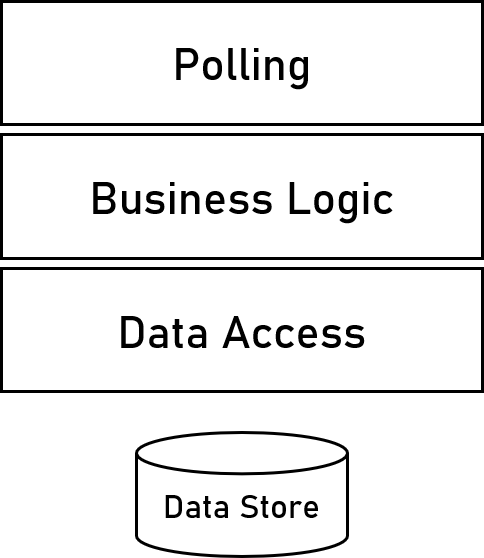
Great, we’ve chosen the processor and the operational database, but there is another one – the archive. What are the main requirements here? We need excellent support for a considerable amount of data,

infrequent retrieval, and we do not care about performance. Last but not least – we want to store the data as cheaply as possible.

What is your suggestion for the archive? Which database will be a perfect fit for the listed requirements?

For this scenario, a great option to consider is using a public cloud. All the major providers have a cheap storage option for archived data. We asked the client whether they use a cloud provider, and their answer was short and sweet “*Amazon Web Services*”. Looking at *AWS’s* pricing for archived storage, we can calculate that our bill will be around $1.00 per TB, which is practically nothing.

After we chose all our technologies, it is time to get back to the processor and design its layers. The 3-layer architecture looks perfect:



You are on your own for the rest of this module.

Pause the book, take a quick break, and answer the following questions as a real architect:

Is your application stateless?

How would you handle redundancy?

How would you handle logging and monitoring?

Do you need caching mechanisms?

What kind of exception handling strategies are you going to implement?

What developer instructions are you going to suggest?

I believe you have excellent answers to these questions! Good job! Now, let’s continue with the *Telemetry Viewer*!

## Designing The Telemetry Viewer

Our next module is a straightforward one. You should be able to design it yourself as you should be very familiar with the process.

Here are the functional requirements for the *Telemetry Viewer*:

* Allows end-users to query telemetry data.
* Displays it in real-time.

Note that the module does not deal with analyzing and reporting. These responsibilities are for the *BI Application*.

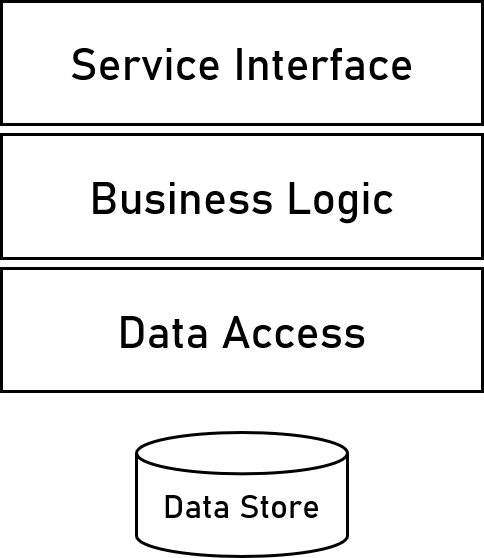
You will not receive any detailed instructions here, so try to independently create the whole module architecture without the book’s help.

The possible solution follows.

The *Telemetry Viewer* is a classic web server application.

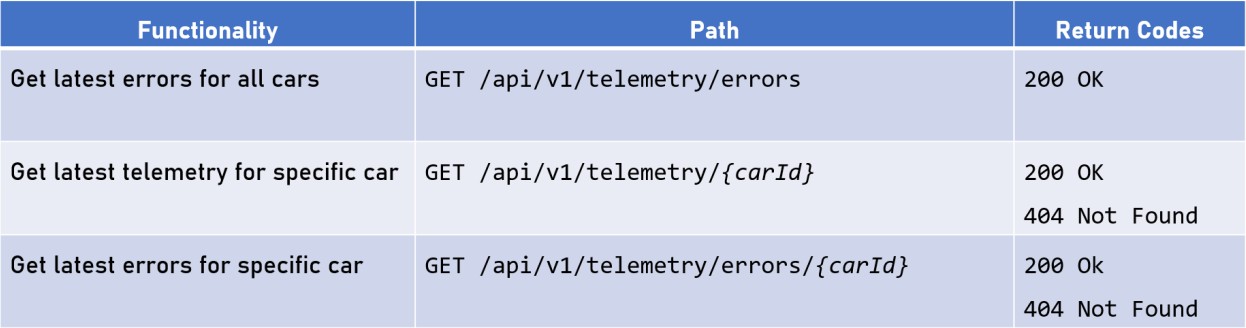
Since it visualizes data in real-time, we will need both back-end and front-end technologies. The back-end is relatively straightforward – it should be *Node.js* because we are already using it in the system. As for the front-end – we need to ask the client’s preferences. They chose *React,* but any other popular framework will be sufficient – *Angular*, *Vue.js*, or *Svelte*, for example.

The 3-layer architecture is a good fit for this module:



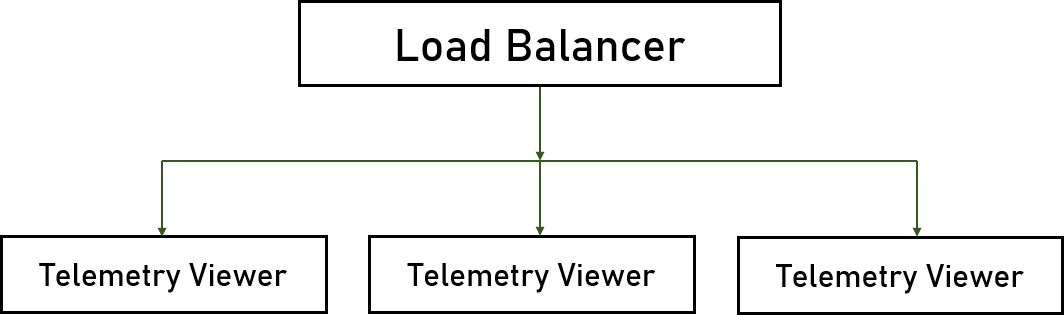
There is a *REST API* involved, so we need to design it. These are the most useful endpoints, based on the business requirements:

* Get the latest errors for all cars.
* Get the latest telemetry for a specific car.
* Get the latest errors for a specific car. Here is the API specification:



You may notice that all endpoints are read-only. There are not endpoints for updating the data because it comes from the cars. We only want to visualize it to the end-users.

Finally, we need to design cross-cutting concerns. Here is the redundancy diagram. Nothing new here:



All the other module attributes are yours to consider. Coming up next – the *BI Application*!

## Designing The BI Application

The last module for the *Bradva Cars* system is the *Data Warehouse*

and the *BI Application*.

Here are the functional requirements:

* Analyzes telemetry data.
* Displays custom reports about the data, trends, forecasts, etc.
  + How many cars did break during the last month?
  + What is the total distance the cars drove?
  + And many other important business questions.

We must have a very comprehensive solution to answer all these questions.

But what is the application type of this solution? The answer is interesting – it doesn’t matter because *BI* applications are always based on an existing tool. We don’t develop business intelligence tools for the same reason because we don’t build web servers from scratch.

Some of the popular *BI* tools are *Power BI, Tableau, QlikView,* and others. How do we pick one? Well, designing such solutions is actually not part of the architect’s job. We should always discuss the topic with a *BI* expert.

It is our job to talk with the customer to involve a business intelligence employee or consultant.

## Final Architecture Diagrams

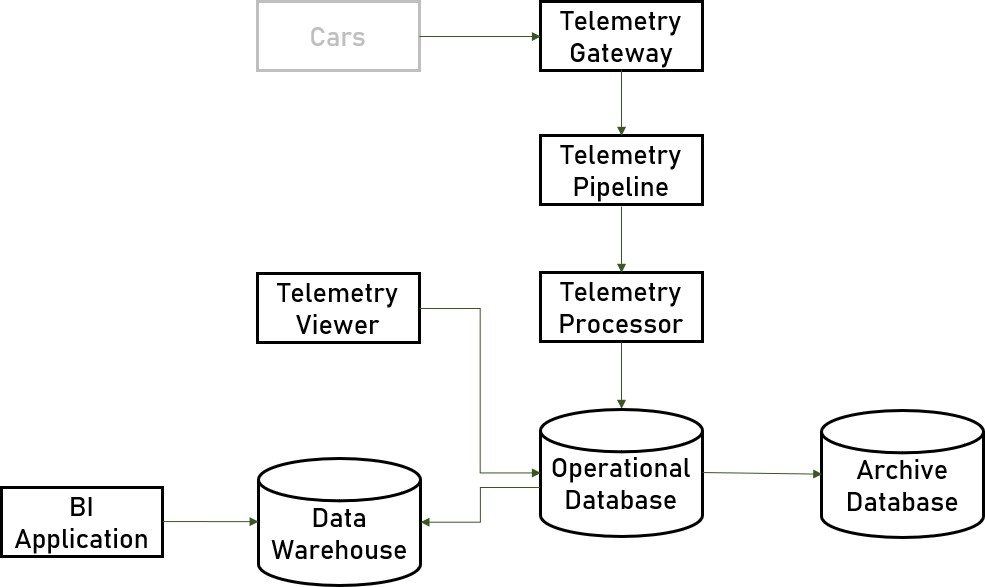
Before we finish the solution for *Bradva Cars*, you have some questions to answer.

What should we do in terms of infrastructure?

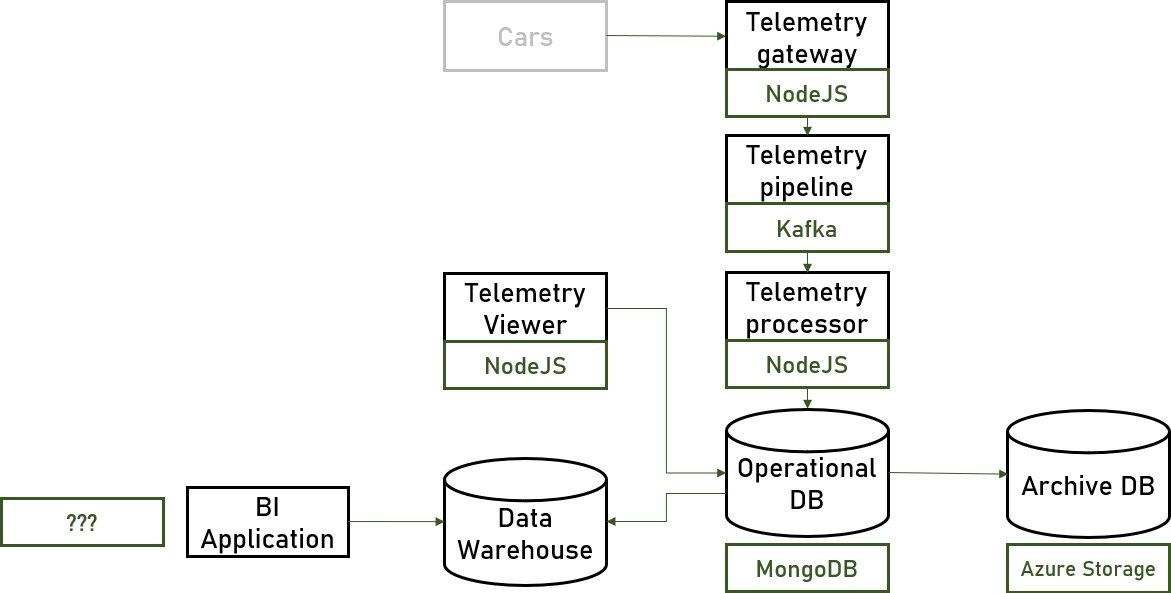
Are we going to include container and orchestration in the system? Do we need a CI/CD pipeline?

After you are ready, we can conclude with the architecture diagrams.

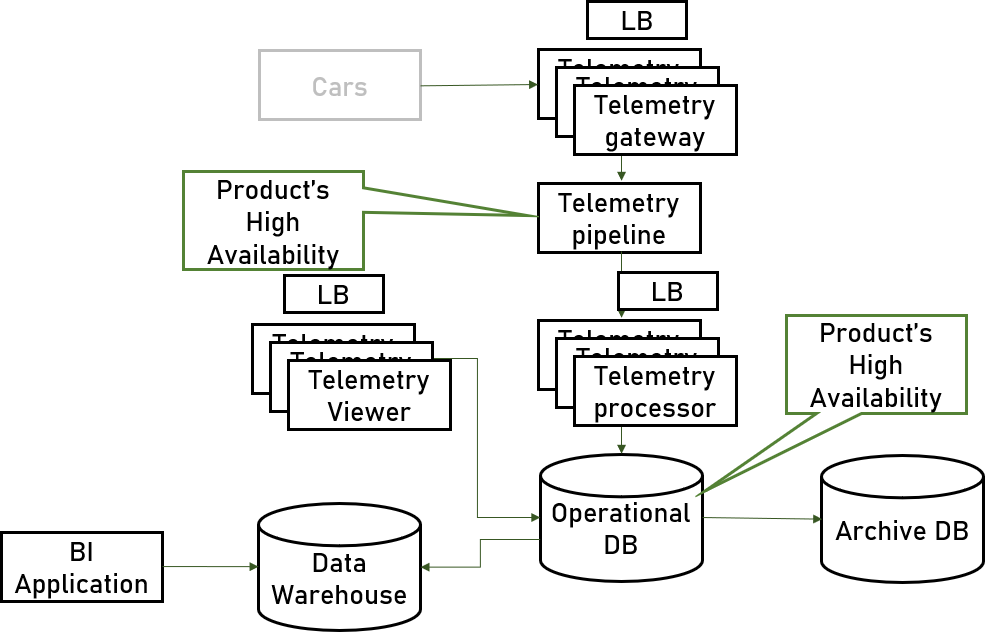
*Logical Diagram*



*Technical Diagram*



*Physical Diagram*



We are ready with *Bradva Car’s* data-heavy application! Perfect!

# System #3 – eTorbichka

## Application Introduction

*eTorbichka* is a grocery collection service. It allows customers to create shopping lists, which eTorbichka delivers to an address. Their business is available worldwide and has great success. *eTorbichka’s* employees have dedicated tablets that display the pending orders. They have the option to mark an item in the list as either collected or unavailable if the product is not in stock in the grocery store.

*eTorbichka* hired you to design the system’s collection side – the one used by the company’s employees. The customer portal is already implemented.

## System Requirements

Like with the previous projects, we need functional and non-functional requirements. Here is what we already know from *eTorbichka*:

* Web-based application.
* Tablets receive lists to be collected.
* Employees can mark items as collected or unavailable.
* When a collection is processed, we must transfer it to a payment engine.
* Offline support is required.

Pause a the book and think about the questions you would ask *eTorbichka*. What are the most important non-functional requirements for this system?

Here there are:

* *“How many concurrent users do you expect?”*
  + 200 concurrent users.
* *“How many lists will be processed per day?”*
  + 10,000 lists / day.
* *“What is the average size of a shopping list?”*
  + The average size of a shopping list is 500 KB.
* *“What is the desired SLA?”*
  + The highest possible.
* *“How do lists arrive from the other system?”*
  + Lists come through a message queue.

Try to calculate the data volume of the system by yourself. It is an easy task, but nevertheless – we have ~2TB per year.

The data volume seems to be quite manageable for this project.

Good. Let’s analyze the final non-functional requirements of the

*eTorbichka* system:

* 200 concurrent users.
* 10,000 lists per day.
* 4 TB yearly volume.
* High *SLA*.
* Offline support.

Sounds interesting. Let’s now identify the system’s key scenarios and

map its main modules!

## Identifying Modules

Now, pause the book for a minute and try to identify the key system modules on your own:

* How many different modules are you going to add to your solution? Define them based on the essential requirements.
* What are the main challenges of the system, and how are you going to handle them?
* How are you going to implement the data storage?
* How are you going to connect the modules? How are they going to communicate with each other?
* Answer carefully to all the questions above and try to create a diagram for your architecture.

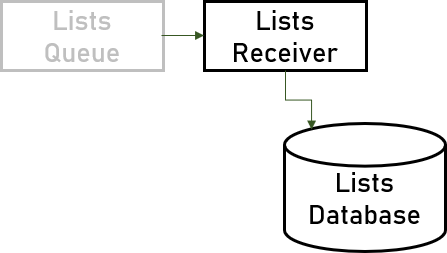
After you are ready, you can continue reading to find one of the possible solutions.

Based on the requirements, these are our key scenarios:

* Employees have tablets.
* Offline support.
* Retrieve lists.
* Mark Items.
* Export list to the payment engine.

First, we need to put the *Lists Queue* module on our architecture. Even though that specific part of the software is already implemented, it is the source of our system’s data.

The next module is the one that receives the lists from the queue. Let’s name it *List Receiver*. This component is also responsible for storing the list records in a database:

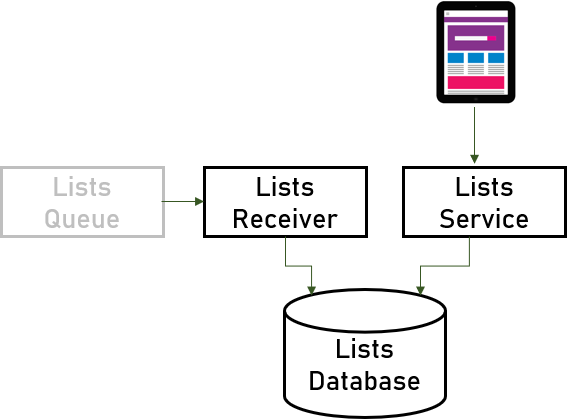


The next module will be responsible for retrieving the lists from the database, updating their state, and exporting the data. Let’s name it *Lists Service*.

But why do we separate the receiver and the handler? They could work together in a single module. The answer is simple yet important

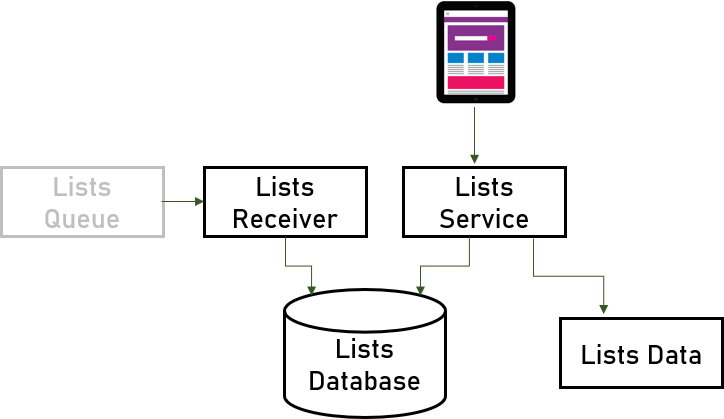
- because we want the service to operate independently of the data source. By separating the responsibility between two modules, we successfully achieve loose coupling. In the future, the list records may come from a database or an *API*. Thus, we are making sure our architecture can be easily adapted.

The next module is the tablet. It is a front-end part of the system, but it has a significant role, so we need to include it. Besides, the tablet needs to support the required offline mode:



Our last module is the *List Data*. The data is generated by the *Lists Service,* and then it is exported to the external payments engine.

Good job! We designed all modules in our system:



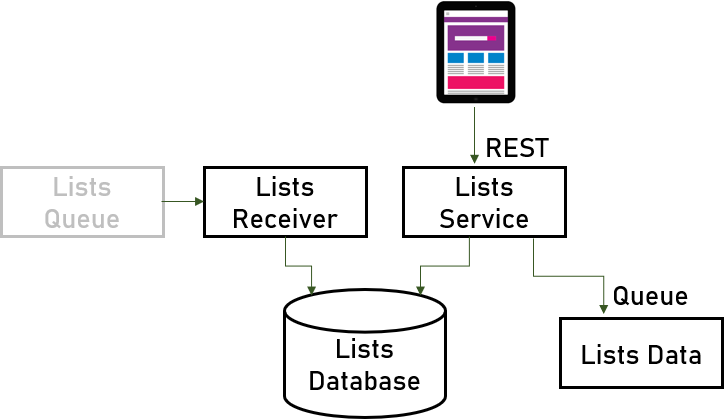
It’s time for messaging. Once again, we need to discuss how these modules will communicate with each other.

What do you think? Think for a moment and design the architecture’s communication.

We have two communication channels that are in our jurisdiction:

* The tablet and the *Lists Service* – they should use standard web protocols – *HTTP* and *REST*.
* The *Lists Service* and the payment system – our module will publish the *Lists Data* to a queue, read by the external module.

Here is a communication diagram of our system:



Looks good. It’s time to start designing our modules. The first one is the *Lists Receiver*!

## Designing The Lists Receiver

A straightforward module. Its functional requirements are:

* Receives shopping lists from the queue.
* Stores the lists in the database.

As usual, we start with the application type and the technology stack. It is your turn now:

* Which application types are the best possible options for this module?
* What will be the best technology stack here?

After answering the questions above, you can continue reading.

As with previous similar examples, we have a long-running task here, so that the perfect choice will be either a console application or a service.

Next is our technology stack. Before we pick it, we need to analyze these considerations:

* Should be able to connect to a queue.
* Should be able to work with a database.

Well, we are not asking for a lot…

Let’s ask *eTorbichka* for help. Their answer is:

*“We’re basically a Java shop, and our database of choice is MySQL.”*

*Java* is a perfect fit for these tasks. It is a mature platform with a lot of capabilities.

But what about the database?

Our shopping list models are structured, and *MySQL* works great with relational data. In this aspect, the customer’s data store technology will be sufficient. Unfortunately, the 2 TB per year volume is a bit too much and cannot be easily handled.

We need to find another solution. A good option here is partitioning.

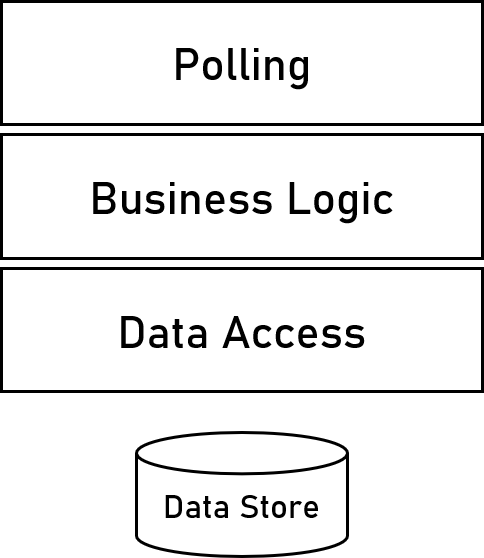
Partitioning allows us to separate the data into groups based on one or more columns. In our case, we can partition the records by their month of creation. We will have a partition for January, for February, and so on.

This way, we allow the database to handle a significant amount of data but with small chunks instead of one big table.

*MySQL* supports partitioning, so there is no reason the change it with another technology, especially since the customer is already using it.

Can you design the module’s architecture by yourself?

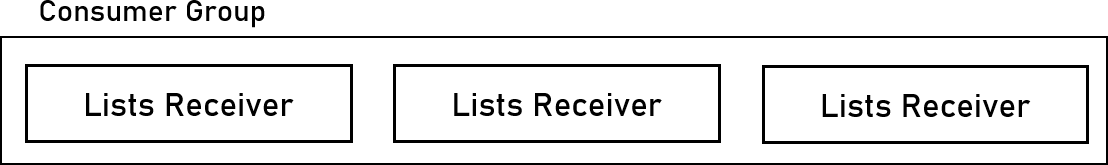
Of course, you can. You are quite a machine after practicing with this book. But for the sake of completeness, here it is:



A classic 3-layer architecture as usual. You can’t go wrong with it! Always chose it as a starting point of your modules’ architectures.

Let’s talk about the redundancy of the module. Here we can leverage a technique called “consumer group”. Most advanced queue implementations support it. The pattern is simple – we define a group of receivers, and the queue delivers a message to only a single one of them. The message broker will do a “round-robin” algorithm between every consumer in the group. If a consumer is not online, the queue will simply send the message to another one.

Here is the diagram of this pattern:



We are pretty much done with this module.

Think about all the cross-cutting concerns and design them accordingly.

It’s time for the *Lists Service*!

## Designing The Lists Service

The *Lists Service* functional requirements are:

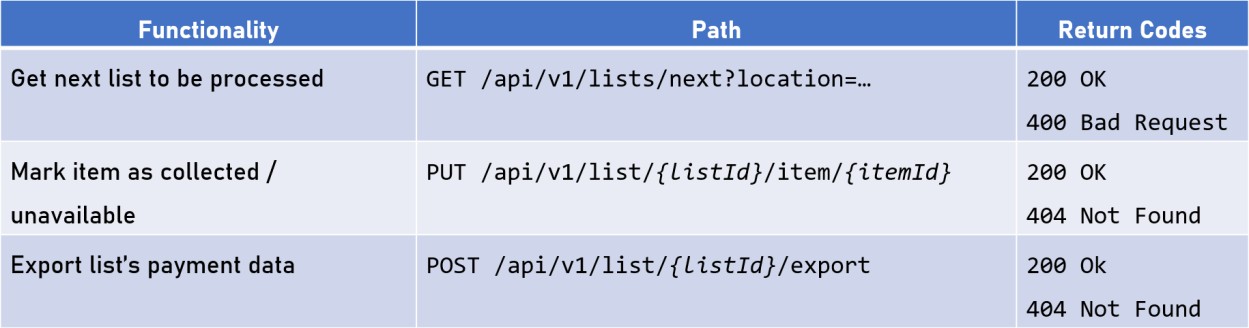
* Allows employees to query lists.
* Marks items in a list as collected or unavailable.
* Exports payment data.

As usual, we start with the application type and the technology stack.

The *Lists Service* module is completely standard. You are entirely designing it on your own. Even a sample solution will not be provided. Consider everything you have practices so far in this guidebook.

Ok, there will be a little hint. Here is a possible solution to the

module’s *API*:



I am sure you did a great job architecting the *Lists Service*. Now, let’s design the tablet’s front-end!

## Designing The Tablet’s Front-End

These are the functional requirements of the tablet’s front-end:

* Displays shopping lists.
* Marks items as unavailable or collected.
* Sends lists to the payment system.
* Supports offline mode.

Let’s choose the application type. There are two options available here solely because of the offline mode support. We cannot use a web application because it requires an *Internet* connection. We also have a *UI* for the *eTorbichka’s* employees. Therefore, our choice is between a desktop or a mobile application.

We need to compare the two options:

* *Desktop*
  + Requires a *Windows*-based machine.
  + Allows us to utilize other applications on the device.
  + Supports all OS functionalities.
  + Requires setup.
* *Mobile*
  + Web-based mobile application.
  + Limited functionality.
  + Cannot use other applications on the device.
  + Fully compatible with all kinds of phones, tablets, etc.
  + No setup.
  + Cheaper hardware.

For our scenario, the web-based mobile application is a more suitable option. We need to choose a technology. There are various alternatives out there, but one of the best supporting offline mode is

*React Native*. It is effortless to learn and is very similar main *React*

library used by many *JavaScript* developers.

We are done with this module. Let’s consider the queue and the *Lists Data*.

## Designing The Lists Data Queue

Which queue would you choose for this task?

The answer is simple enough. There is already a queue in the *eTorbichka’s* technology stack. We should stick to the same one. Consider the design done! All we need to do are the final architecture diagrams.

Caching and final architecture diagrams!!!

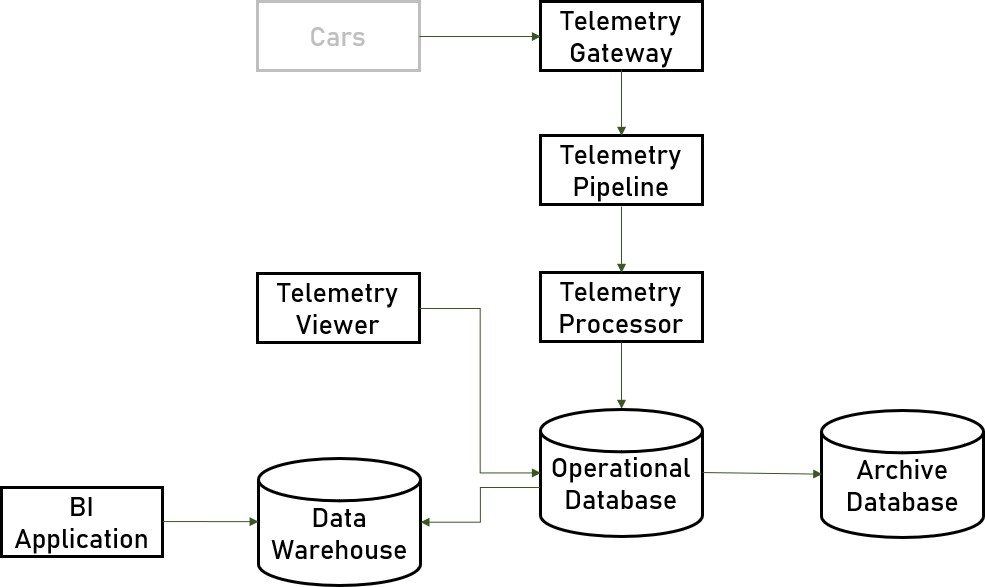
## Final Architecture Diagrams

Before we finish the solution, you have some questions to answer. What should we do in terms of infrastructure?

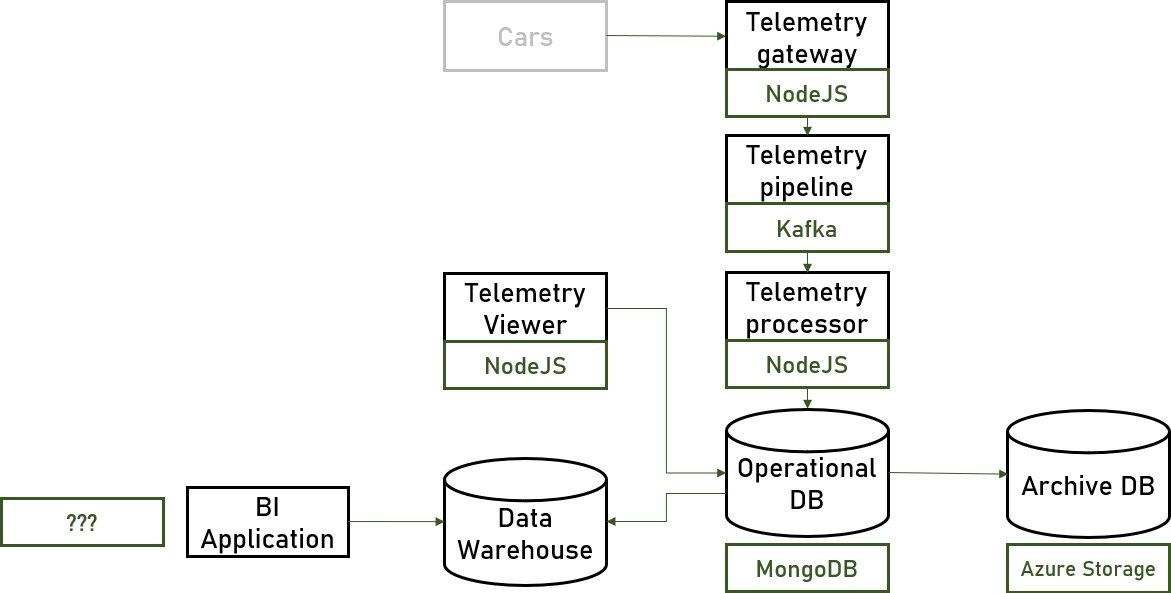
Are we going to include container and orchestration in the system? Do we need a CI/CD pipeline?

After you are ready, we can conclude with the architecture diagrams.

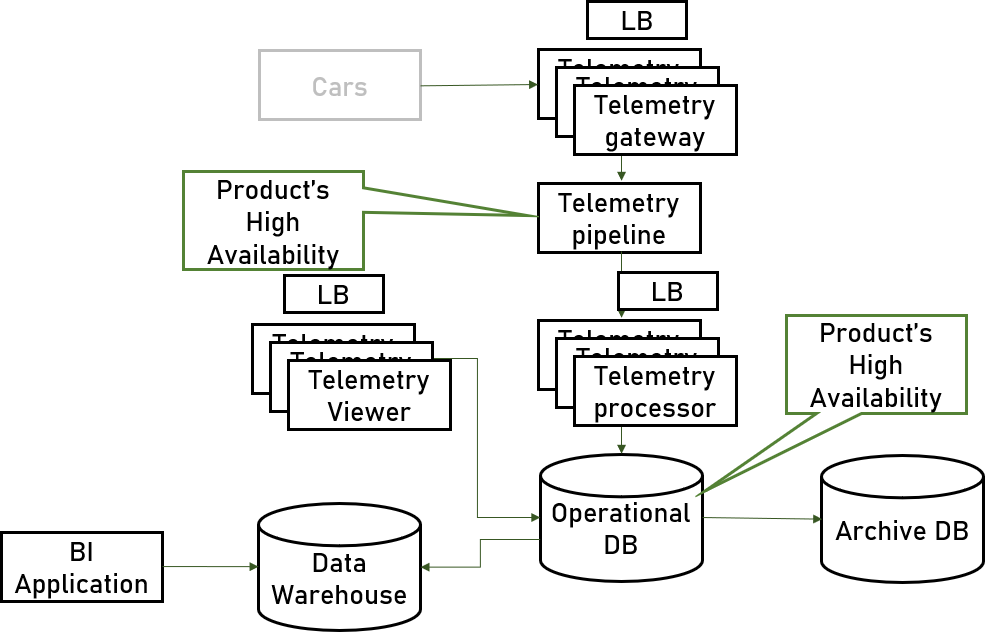
*Logical Diagram*



*Technical Diagram*



*Physical Diagram*



To be continued… Two more projects coming soon! Stay tuned! ◆