Designing Software Architectures

For Scalability, Availability, and Manageability

Practical Step-by-Step Guidebook

Foreword

Welcome to the software architecture practical guidebook. Here, you will find various real-world case studies related to software solutions and design. All of them are quite interesting, challenging, and contain a little twist.

We are going to cover and calculate performance metrics. We are going to provide and support redundancy. We are going to consider a lot of seemingly hidden requirements. And finally – we are going to have some decent architectural fun!

Sounds great, but why do we need case studies and practical examples? What is the point of them, and what are we going to learn exactly? Well, the goal is simple. I want you to become way better at software design and initial solution considerations - a great architect.

And great architects know how to deal with various types of systems.

Designing a proper web application is just the start. What about systems with lots of incoming data? Or systems with an absolute reliability requirement? Or maybe a system working with sensitive information?

We are going to cover all these scenarios and even more.

We will cover multiple types of systems and design their architecture step by step, considering all client requirements.

Most importantly, this book is not theoretical and fictional.

We are going to analyze real-world architectures based on existing production environments. Actual systems for actual applications with actual requirements that people have implemented around the world.

Now, let’s start!

Disclaimer

The real names of the projects behind the case studies were changed for fictional ones to protect the projects' identities. Additionally, various unnecessary details have been removed from their requirements to make the solutions more suitable for this book's educational purpose. Yet, the fundamental goal of the systems and their architectural designs were left entirely intact. Finally, the solutions provided here are perfect for their purpose but are certainly not exclusive. They can be easily replaced by other possible designs that may be perfect for doing the exact same job.

Guidebook Instructions

To get the maximum out of this book, there are practical exercises included. Every time you see an underlined question, pause the book and think about the possible answer. After you are ready with your solution, you may continue reading to understand the author's design thoughts around the architectural concepts.

The Architecture Process

In this section, we will briefly discuss the process of designing a functional software architecture that provides the maximum possible value to the client. Every architect should follow a well-defined roadmap to ensure the system is fast, secure, reliable, and easy to maintain:

- *Understand the system requirements* – high-level tasks of what the system should do - usually provided by a functional analyst.
- *Understand the non-functional requirements* – define the system's technical and service level attributes: load, data volume, concurrent users, and more.
- *Map the architecture modules* – the moving parts, which represent the various tasks of the system. The essential concepts here are their separated and isolated responsibilities as well as the communication between them.
- *Select technology stack* – there are many factors we must consider for selecting the perfect tools for the job. The decision here must be very rational because a wrong technology stack may lead to the whole system's failure.
- *Design the architecture* – the heart of the architect's work. It should result in a system that is fast, secure, reliable, and easy to maintain.
- *Write the architecture document* - describes the whole design process and gives the entire team a full picture of the future system.
- *Support the development team* – the architect must stand right next to the developers and help them with the coding process. There will be dilemmas and arguments. The architecture will change. And not only once!

System #1 – Paper & Son Limited

Application Introduction

Paper & Son Limited is a company that sells paper supplies all over the world. They desperately need a new HR system, which will be responsible for managing:

- Employees
- Salaries
- Vacations
- Payments
- Bonuses
- Promotions

Paper & Son Limited hired you as a consultant to help them design their new and shiny system, so your first order of business should be gathering all necessary requirements.

System Requirements

Before designing the solution, we need to gather:

- Functional requirements – what should the system do?
- Non-functional requirements – what should the system deal with?

Usually, the client should know the initial functional requirements. More prominent companies hire functional analysts to document these. In this case, *Paper & Son Limited* provided the following:

- 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.
- Uses an external payment system.

Now, let's look at the non-functional requirements. The first thing we need to find out is what we already know about these non-functional requirements. The system does not seem complicated at all:

- Classic data-driven application.
- There is not a lot of data moving around.
- There are not a lot of users.
- There is an external system involved.

We need to consider the final requirement seriously because if the payment processor is a legacy system, we may experience potential problems.

So, we figured out what we know already. But what do we need to ask the client before moving on?

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

- Think about all the questions you would ask if you were the actual software architect of this system.
- What parts of the system are still covered in mystery, and you will need to understand before moving on?

After you are ready, you can continue reading to find the answers.

Paper & Son Limited is a big corporation, so we need to ask them how many concurrent users will use the system simultaneously. This

evaluation is essential for estimating and designing the load requirements for our solution.

The second question is about the number of employees that are going to be managed by the system. If it is a vast number, then our data volume may become a burden.

The final question is about the payment processor. We need to gather as many details as possible every time we are dealing with external systems. Is it a web interface? Or a file system? How do we pass the data to it?

In summary, these are the questions we need to ask:

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

And these are the answers we got from the clients:

- 10 concurrent users. It should not be a problem.
- 250 employees. Not an issue.
- Payment processor:
    - Legacy system written in *C++*. It is getting complicated…
    - Hosted in the company's server farm. At least we do need to work with external networks.
    - Input – only with files. Bad news.
    - A file is sent once a month. Even more bad news.

Apparently, the only way we can send any data to the legacy payment system is by using files once a month. In other words, our system must produce and send a payment file monthly. It must contain all the data about the transactions that must be sent to every employee's bank account.

It looks like this file is working with a considerable load of data but let's not make any assumptions and calculate everything precisely.

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. *Paper & Son Limited* expects 500 employees in five years. The total storage required is:

51 MB x 500 Employees = 25.5 GB

This number is tiny by today's hardware standards, but we will need to consider where to store the documents during the technology stack selection.

A final consideration of the system is the *Service Level Agreement* (*SLA*). Since it is an *HR* management system, the client says it is not critical to have downtimes here and there.

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

- 10 concurrent users.
- Manages 500 employees.
- 25.5 GB data volume forecast.
    - Relational & unstructured.
- Not a mission-critical system.
- File-based external payment system.

Great. We are 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:

- How many different modules are you going to add to your solution?
- 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 our requirements, we have these initial entities – *Employee*, *Vacation*, *Salary*. The latter will include bonuses and promotions. Additionally, we have an interface to the external payment system.

The best possible scenario is to have a single service or a single module for a single entity. We want a modular and maintainable system. If we introduce multiple entities to a module, we will lose these attributes.

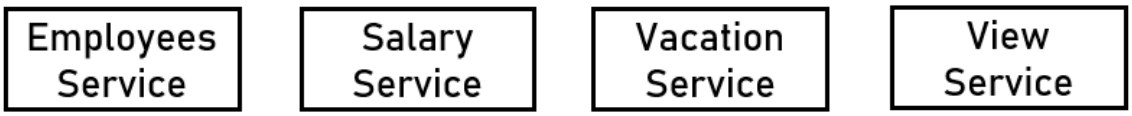
Based on our entities, we would define the following modules to serve our tasks:

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

These modules are logically separated. We are not talking about four different web servers here. Considering the concurrent user requirements, the developers can safely implement all services in a single web application. Architecturally, however, we need to map them as separate entities.

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:



We will also need a *Payment Interface* to work with the external payment system:

We still haven't mentioned one of the essential modules in the system – the data store.

One of the most common questions in such scenarios is whether we need a single database or four databases – one for each service. And that is an excellent question. This is why it is so common.

The answer here is simple – there is no rule of thumb. It depends on the system's requirements and the shareability of its data.

In our situation, we have employees connected to their salaries and vacations. Our data is shared. If we decide to split it, we are going to implement the microservices architectural pattern. It provides scalability at the cost of additional complexity.

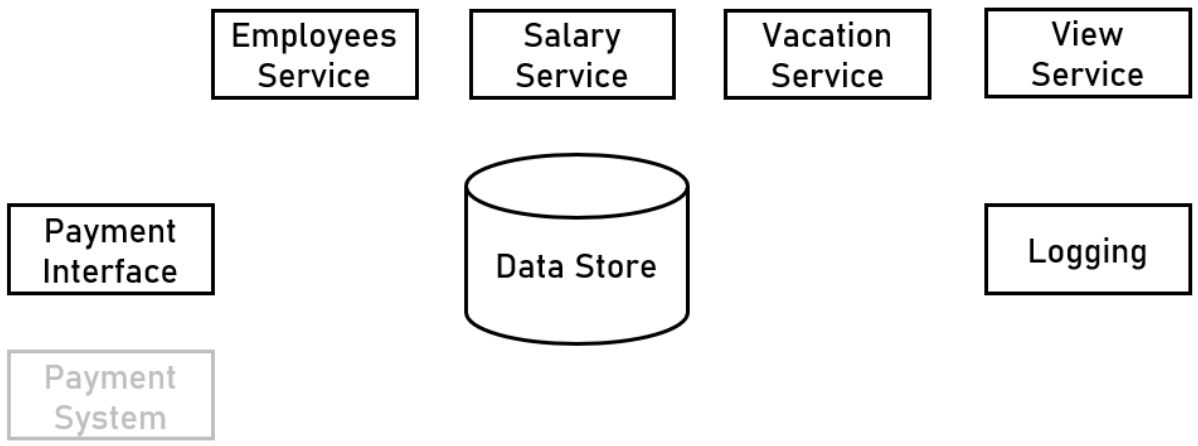
Is that complexity justified? With no more than 10 concurrent users – definitely not!

Based on our requirements, a shared database seems to be a perfect choice.

Finally, we need a logging service to collect logs and show what is happening in the system.

As a result, our architecture will have six distinct modules.

Here there are:



The next consideration we need to discuss is how exactly these modules will communicate with each other. It is time for the messaging phase of our design. We need to decide what is the contract for each service and the transport mechanisms behind it.

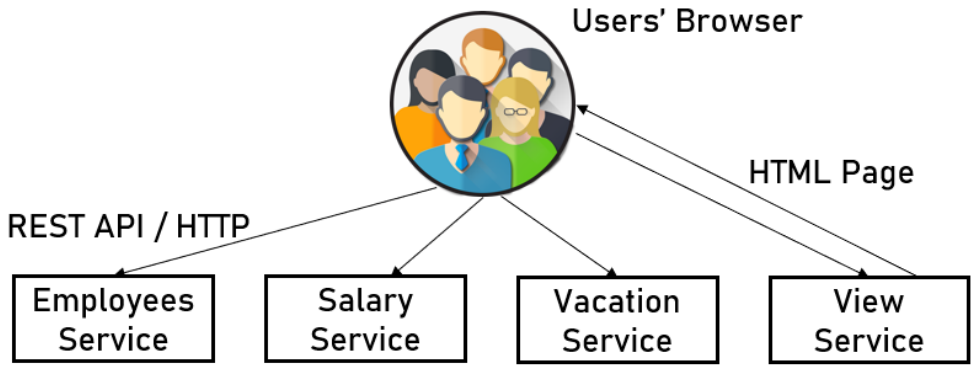
Before we decide on the messaging protocol, we will need to know who will use the different services.

The top four services – *Employees*, *Salary*, *Vacation*, and *View* – will be called by the user's web browser.

In more detail, the browser is going first to call the *View Service* to retrieve the *HTML*, *CSS*, and *JavaScript*. It will render a web page, and then it will request data from the other three services.

The standards for the above scenario is *HTTP* and *REST API*. The implementation will be quick and easy because all popular technologies support them.

Our diagram so far:



All these services need to send log data to the *Logging Service*.

What is the best communication protocol here? Why?

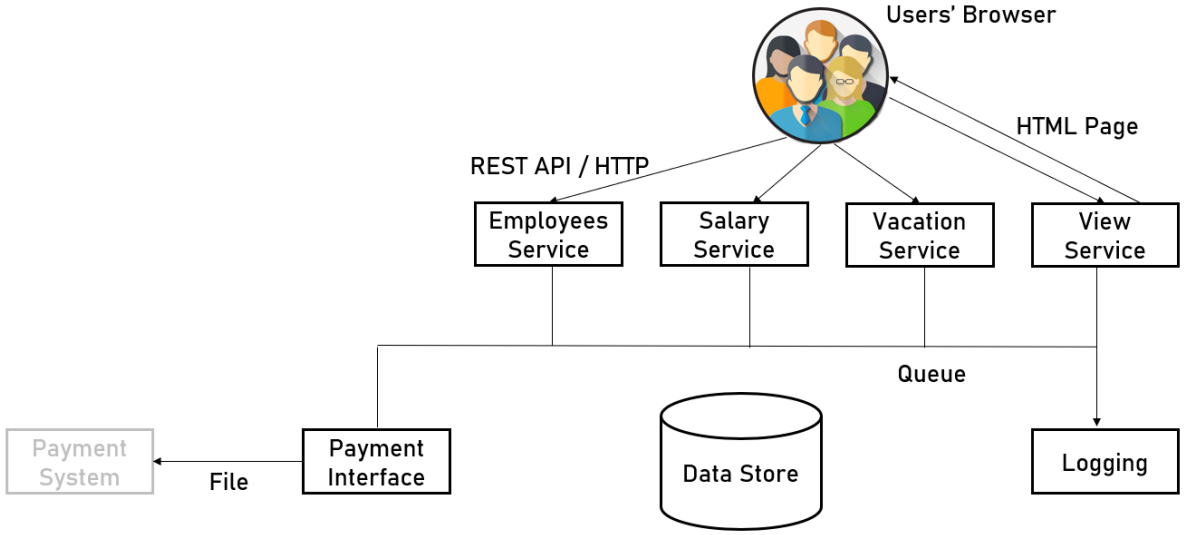
Think about the answer before continuing.

The protocol we should use here is asynchronous - a queue. The services do not need to wait for any response. They just need to "fire and forget" the log data. The queue is the perfect mechanism for that requirement.

The last consideration is about the payment service. We do not have a lot of choices available. Our *Payment Interface* needs to create a file in some shared network. Then the external system can pull it for processing.

We have finalized our solution by identifying the different modules and specifying the communication protocols between them.

Here is our final diagram:



The module mapping part of the process is complete. Now, it is time to design each part of the architecture one by one!

Designing The Logging Service

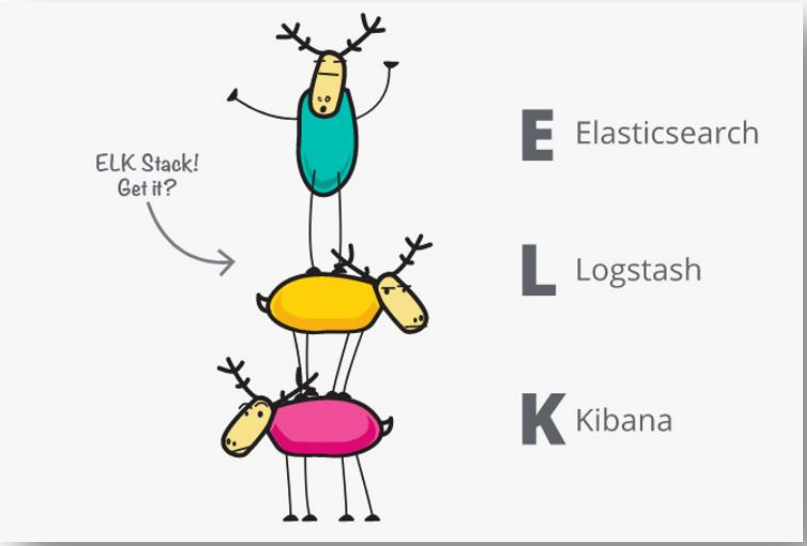
The first module we will design is the *Logging Service* because it serves a critical task. It logs all activities in the system and locates any problems or malicious user behaviors that may arise. On top of that, every other service uses it directly.

There are some questions we need to ask ourselves before continuing our work:

- Is there an existing logging mechanism in the company?
- If the answer to the first question is negative – should we develop our own tool or use a third-party alternative?

After asking the customer, we now know the company doesn't use a centralized logging mechanism. Each one of their applications has its own solution for the cross-cutting concern.

As for the second question – a well-known third-party solution is the *ELK* stack – *ElasticSearch*, *Logstash*, and *Kibana*:



Here is what the stack includes:

- *ElasticSearch* – a robust and indexed data store.
- *Logstash* - imports log data from many sources.
- *Kibana* – a great dashboard with lots of filter capabilities.

The *ELK* stack does a beautiful job, and many organizations use it.

Now, pause the book to test your architect skills and consider the following questions:

- Should we use the *ELK* stack for this solution?
- Are there any adverse consequences of including the stack in our architecture?
- Which would cost the company more – building its own tool or using a third-party one?

Think about the answers before continuing.

Here are the cons of the *ELK* stack:

- Each different product in the stack requires technical support.
- The tools are quite complicated to install and setup.
- The *ELK* is designed mainly for large, data-intensive systems.

The rule of thumb with these scenarios is that we should always aim to include well-supported third-party tools because they save us a lot of time and effort. However, we cannot blindly have enterprise-level software in a medium-sized system like the one we are currently designing.

After a quick look at our non-functional requirements, we can conclude that the solution we are building will not have a substantial data load. The *ELK* stack is not a simple library to "plug-and-play" in our architecture. For this reason, we will develop our own logging mechanism.

Here are the steps required:

- Choose the application type.
- Decide the technology stack.
- Design the architecture of the module.

It is time for you to pause the book again and design the service by yourself.

- What is the purpose of this service?
- What are the best technologies you can use?
- What inner layers and components are you going to use?
- Try to create a diagram for your solution.

You can continue after you are ready.

Let's start with the application type but first to define the main tasks of the module:

- Read log records from a queue.
- Validate and store these records in a data store.

The *Logging Service* is not relying on a request/response mechanism and serves a long-running task. A console or a background service application will be a perfect choice here. But which one should it be?

Well, both solutions are great, and it does not actually matter. In this humble author's opinion, we should use a background service because it will be easier to restart in case of a failure. The operating systems have automatic configurations for such cases.

Our application type will be a service. What about the technology stack? In what language will we write the module's code, and what will be our data store?

Let's start with the code. The module's job is simple enough. There are very few platforms out there that are not able to handle such tasks.

We have many options to choose from, so part of the decision here should come from our client:

"What is the current knowledge and skill set of your team?"

The answer that we've got is:

"We're using the Microsoft stack throughout the company. We are experts in .NET and SQL Server."

This news is excellent for us.

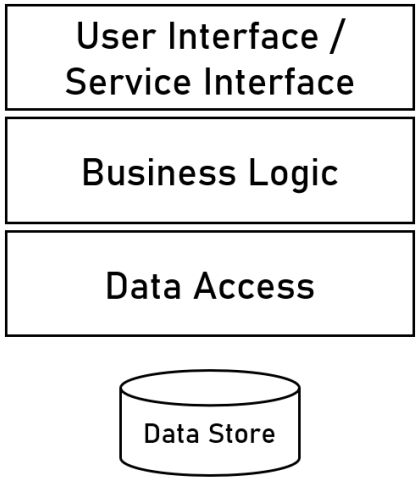
.NET is a compelling platform with one of the best developer tools on the planet. *SQL Server* is also an outstanding relational database used by a lot of organizations. There is no reason not to use these technologies in our case.

Let's now continue to the module's architecture.

The main rule to follow for every initial module architecture is:

- If the module is simple enough – start with the classic 3-layer architecture:
  - *User Interface / Service Interface* – responsible for exposing the data in a consumable manner.
  - *Business Logic* – responsible for data calculation and processing based on the business domain.
  - *Data Access* – responsible for storing and retrieving records to and from the database.
- If the module contains a very complicated business logic – work with domain-driven design and clean architecture layers:
  - *Entities*, *Value Objects*, and *Aggregates* for mapping our business objects.
  - *Factory* and *Repository* design patterns as an *Anti-Corruption Layer*.
  - *CQRS* and *Mediator* design patterns for separating responsibilities.
  - *Domain*, *Application*, *Infrastructure*, and *Presentation* layers.

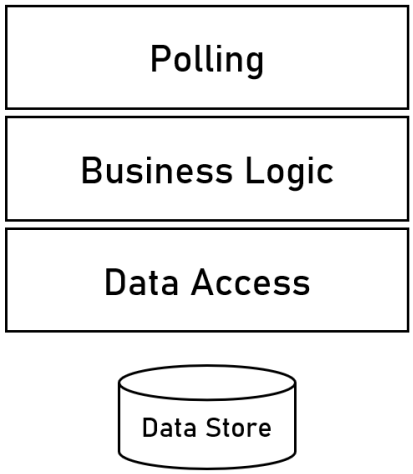
The *Logging Service* is quite simple, so let's start with the 3-layer approach. Here is the initial default design:



Analyzing the layers, we can easily conclude the top layer is irrelevant for the logging module. Simple enough, it does not have a user interface or a web *API*. Our top layer is just a "pulling" layer, responsible for fetching data from a queue. Then it passes that data to the *Business Logic* layer for validation. Finally, we have the *Data Access* layer, which stores the log records in our database.

In conclusion, we started with the classic 3-layer approach and adjusted it to fit our needs.

Here is the updated diagram:



Our next topic discussion is the module's redundancy.

But what exactly is redundancy? Redundancy defines how the service can still function after a crash or a critical error.

As we mentioned earlier – the *Logging Service* is quite crucial for the system. We should design excellent redundancy for it.

We currently have only one instance of the service. If something happens to it and it stops working, we will have a problem. The easiest solution is to add another instance to save the day if the first one is not online.

Usually, the standard practice for redundancy with multiple instances is to provide not two but three replications. However, considering our concurrent users requirement and the small amount of data moving, we can simplify the solution and not over-engineer it. Two *Logging Service* instances will be absolutely acceptable for such a "quiet" system. However, we have a problem…

Can you recognize and solve it without reading the provided solution in the next paragraphs?

Here is the problem - how will we avoid duplicate reads if two instances are pulling records from the queue simultaneously?

Now that you know the problem – can you solve it elegantly?

Stop reading for a moment, and think about the answer.

The solution is quite simple – both the instances will be up and running, but one of them will be the primary one, actively reading data from the queue. The other will stay silent and will check whether the first one is up and running. If there is a crash, the second instance will see the problem, become the primary one, and start pulling data from the queue. We can configure the operational system to restart automatically crashed services. After it is back online, the first instance can become the "silent" one, and just health check the other.

The above solution is known as the *Active-Active* architectural pattern.

Let's now think about whether we need any system-wide attributes in this particular module:

- It is stateless and scalable. We do not store any data in memory.
- It is redundant. We made sure we cover a critical crash scenario.
- Do we need a cache? No, not at all.
- Exception handling strategy - catch, log, and retry 3 times is an excellent way to handle this module's errors. Retrow and crash if 3 consequent retries fail.

We are now ready with the *Logging Service* and its design. Good job!

As a final note, we can recommend additional developer instructions for the team in our architecture guidelines:

- Usage of dependency injection between the layers. It will promote loose coupling in the system. We can even recommend the technology's standard. In this particular module, it is "*Microsoft.Extensions.DependencyInjection*".
- Usage of an *ORM* in the *Data Access* layer. The *.NET* standard is *Entity Framework*. There is no need for the *Repository* and *Unit of Work* design patterns because *Entity Framework* already implements them.

Great. Now, let's move on to the *View Service*!

Designing The View Service

It is time to analyze the requirements for the *View Service*:

- Handle requests from the end-users' browsers.
- Return static files – *HTML*, *CSS*, and *JavaScript*.

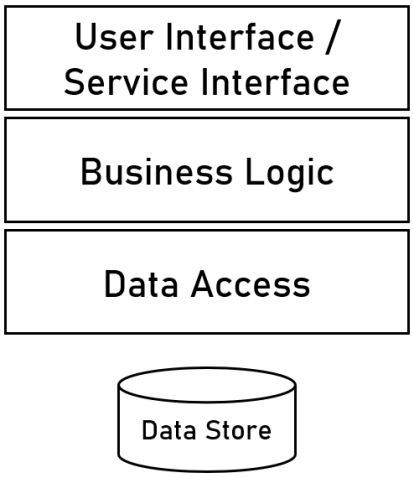
This service serves the browser's static content, which will request or update data from other modules.

The application type is more than evident here – we need a web server.

What about the technology stack? We already chose *.NET* and *SQL Server* for the *Logging Service*. The *View Service* does not have any data store; it is purely code, so we need to consider only the .NET platform. Do we need to change the technology? .NET has excellent support for web applications, and it is one of the leading platforms

worldwide. We need to have an extremely valid reason to introduce new technology in the solution's stack. And we don't have one.

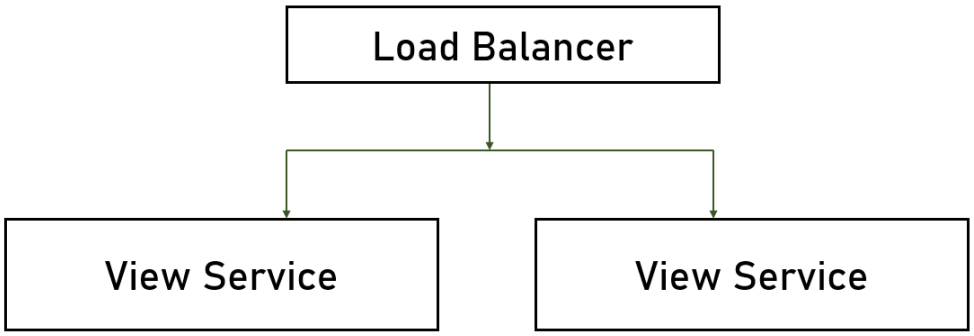
Let's move to the module's architecture. Once again, we have a simple business case. Therefore, we will start with our initial 3-layer architecture:



Which of the above layers are relevant for our *View Service*? The answer is quite simple. We do not have any business logic, and we do not need a database. The service simply returns files the user is requesting. Thus, we only need the *User Interface* layer. Our architecture here is relatively thin:

Our next consideration for the *View Service* is redundancy. The solution is more straightforward than the Logging Service. In request-response scenarios, a load balancer in front of the instances usually achieves redundancy. The only necessary condition for them is to be

stateless. In this case, we do not need to introduce any code changes to our services. Our design is more than enough:



Before moving on to the next module, we need to consider all cross-cutting concerns. The only one worth mentioning is caching.

Pause the book for a bit and try to answer the following questions:

- Do we need caching of any form?
- Since this service returns only static files, do we need to consider a *CDN*?

Think for a minute and continue reading.

The answer lies in our non-technical requirements. We will only have 10 concurrent users in our system. Any server technology can handle such a small load with modern hardware. Therefore we do not need caching, and indeed – we do not need a *CDN*.

Good job! It's time for our *Employees Service*!

Designing The Employees Service

The requirement for the *Employees Service* is to provide *CRUD* operations on the *Employee* entity. It does not contain any visual responsibilities because the *View Service* covers them all.

Here are the functional requirements:

- Allows end-users to query employees' data.
- Allows actions on the data (CRUD).

The application type is as straightforward as possible – we have a web server exposing a *REST API*.

The technology stack is also the same. Once again, we do not have any reason to change it. Our development team will use the *.NET* platform to implement the *Employees Service*.

Now, let's talk about the database. It is a bit more interesting because we have two types of data regarding the *Employee* entity:

First, we have structured relational data about each employee. We will store it in *SQL Server*. This one is easy.

Second, we have unstructured document data that is attached to each employee. How are we going to handle it?

Pause the book and think for a minute about all the possible solutions.

When we talk about documents and files in general, the usual term is *BLOB* (*Binary Large OBject*). In other words, we need BLOB storage. Let's consider our options:

- *Relational Database* – most databases provide mechanisms to store BLOB data.
- *File System* – store the document in a file and hold a pointer to it in the database.

- *Object Store* – specialized software designed and built for storing BLOBs.
- *Cloud Storage* – a viral mechanism for storing large objects, but it costs additional money.

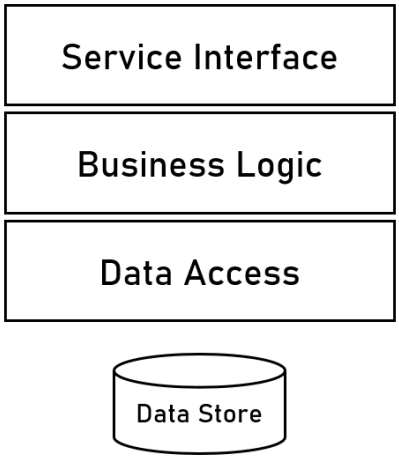
Here is a comparison between these options:

Alternative	Description	Examples	Pros	Cons
Relational Database	Store the document in a specialized column type designed for BLOBs	SQL Server's FILESTREAM, Oracle's BLOB type	Part of the app transaction Part of the DB's backup / DR	Clunky syntax, Limited size
File System	Store the document in a file, and hold a pointer to it in the DB	File System (duh…)	Unlimited size Easy to execute	Not part of transaction, Unmanageable
Object Store	Use special type of store mechanism that specializes in BLOBs	CEPH	Great scale Unlimited size	Complex setup Dedicated knowledge New product in the mix
Cloud Storage	Store the documents in one of the public cloud storage mechanisms	Azure's Storage Account AWS's S3	Great scale Easy to execute	Requires internet connection Cost

We automatically exclude the *File System* and the *Object Store* as possible solutions for our system. The first is too unmanageable in terms of support, and the latter introduces a new complex tool to our technology stack.

The choice here is between our relational database – *SQL Server* – and the cloud. Both options are fine, but let's check our non-functional requirements. Our documents are relatively small – around 1 MB. There is no need to introduce a cloud *SDK* in our toolbox for such files, especially if it increases the system's budget.

Let's continue with the architecture of the *API*. We do not have any complicated business logic, so our initial architecture will be the 3-layered one:



There is no reason to change the layers for this module.

We need to provide the developers team a description of the *REST API* to make their work easier and straightforward.

The *Employees Service API* should contain the following functionalities:

- Get full employee details by *ID.*
- List of employees by parameters.
- Add employee.
- Update employee details.
- Remove employee (with no actual physical deletion).

Additionally, we need to design the documents *API*:

- Add document.
- Remove document.
- Get document.
- Retrieve documents by parameters.

One crucial question to ask ourselves here is whether to extract a separate *Document Handler Service*. Since only the *Employee* entity is working with the documents – there is no need. At least not on an architectural level. Later on, the developers may decide to split the *API*

into multiple controllers or handlers to make the code following the "separation of concerns" pattern.

We need to provide the following in our architecture for each API endpoint:

- *HTTP Verb* – the allowed HTTP method for the endpoint.
- *Path* – the path of the endpoint.
- *Status Codes* – the possible status codes from the endpoint.
- *Data Schema* – request and response data models.

Pause the book and try to design the *Employees Service API* by yourself.

We will not cover the standard *REST API* best practices in this guidebook because it is not part of its scope.

As a reference, here is the solution for the *Employees API*:

Functionality	Path	Return Codes
Get employee details by ID	`GET /api/v1/employee/{id}`	`200 OK` `404 Not Found`
List employees by parameters	`GET /api/v1/employees?name=…&birthdate=…`	`200 OK` `400 Bad Request`
Add employee	`POST /api/v1/employee`	`201 Created` `400 Bad Request`

Update employee details	`PUT /api/v1/employee/{id}`	`200 OK`
		`400 Bad Request`
		`404 Not Found`

Remove employee	`DELETE /api/v1/employee/{id}`	`200 OK` `404 Not Found`

And for the *Documents API*:

Functionality	Path	Return Codes
Add document	`POST /api/v1/employee/{id}/document`	`201 Created` `404 Not Found`
Remove document	`DELETE` `/api/v1/employees/{id}/document/{docid}`	`200 OK` `404 Not Found`
Get document	`GET /api/v1/employees/{id}/document/{docid}`	`200 OK` `404 Not Found`
Retrieve documents for employee	`GET /api/v1/employees/{id}/documents`	`200 OK` `404 Not Found`

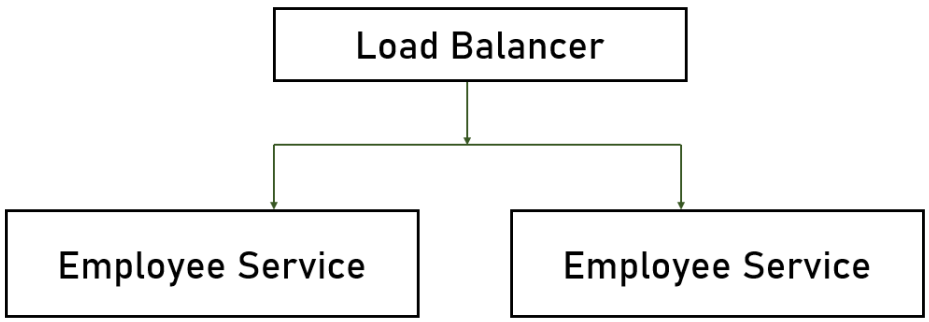
Our next consideration is about redundancy.

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.

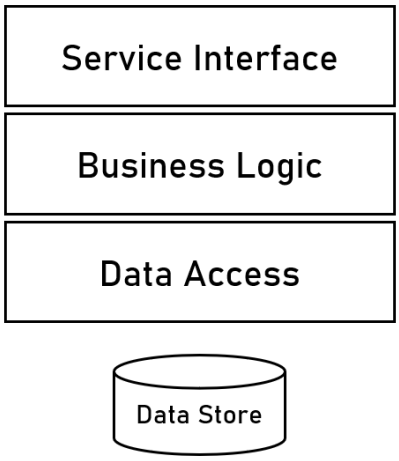
Pause the book. No, actually, stop it altogether!

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:

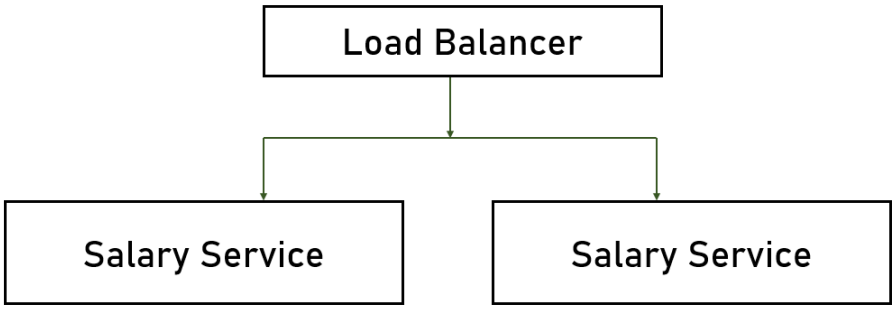
- Add salary request.
- Remove salary request.
- Get salary requests.
- Approve salary request.
- Reject salary request.

Here is the specific breakdown:

Functionality	Path	Return Codes
Add salary request	`POST /api/v1/salaryRequest/`	`200 OK` `400 Bad Request`
Remove salary request	`DELETE /api/v1/salaryRequest/{id}`	`200 OK` `404 Not Found`
Get salary requests	`GET /api/v1/salaryRequests`	`200 OK`
Approve salary request	`POST /api/v1/salaryRequest/{id}/approval`	`200 OK` `404 Not Found`
Reject salary request	`POST /api/v1/salaryRequest/{id}/rejection`	`200 OK` `404 Not Found`

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:

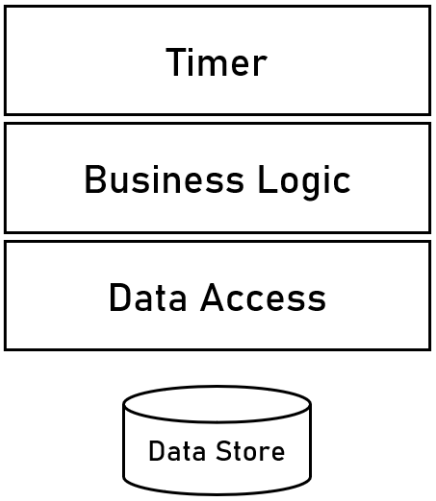
- Queries the database once a month for salary data.
- Passes payment data to the external payment system.

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:

Alternative	Description	Pros
Rabbit MQ	General purpose message-broker engine	Easy to setup Easy to use
Apache Kafka	Stream processing platform	Perfect for data intensive scenarios

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.

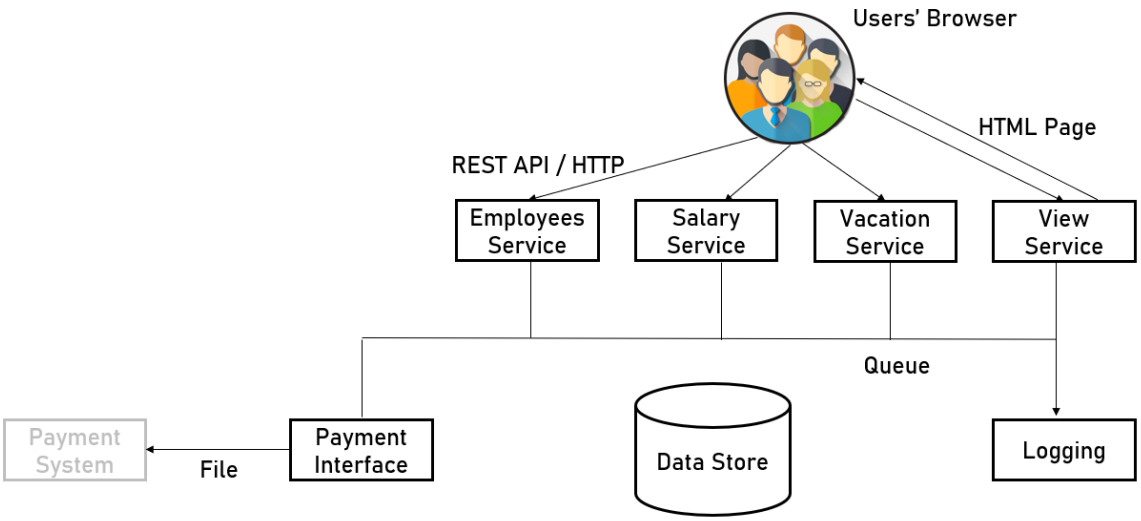
First, we need a logical diagram. It describes the different modules we designed and how they communicate with each other.

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

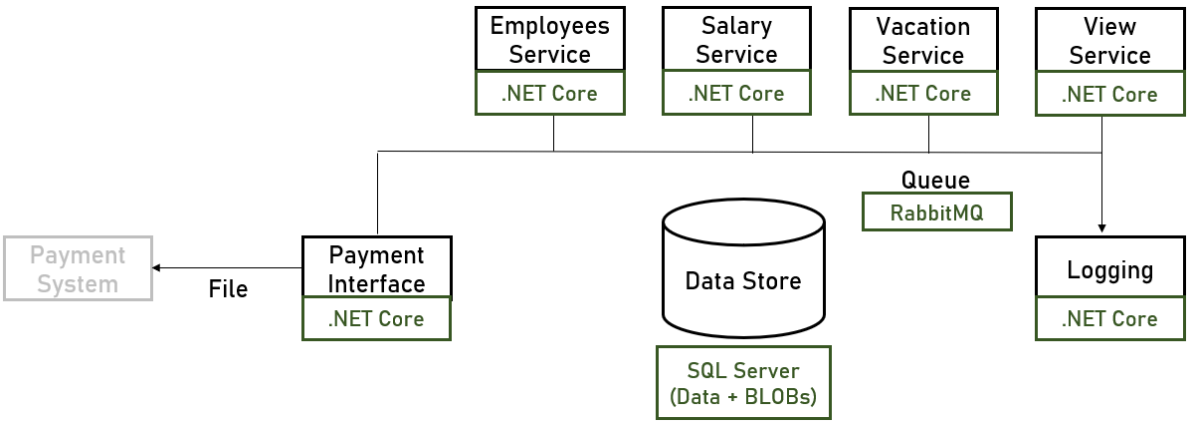
Finally, we have a physical diagram. It depicts the redundancy of each module and how to develop and deploy it on the hardware.

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.

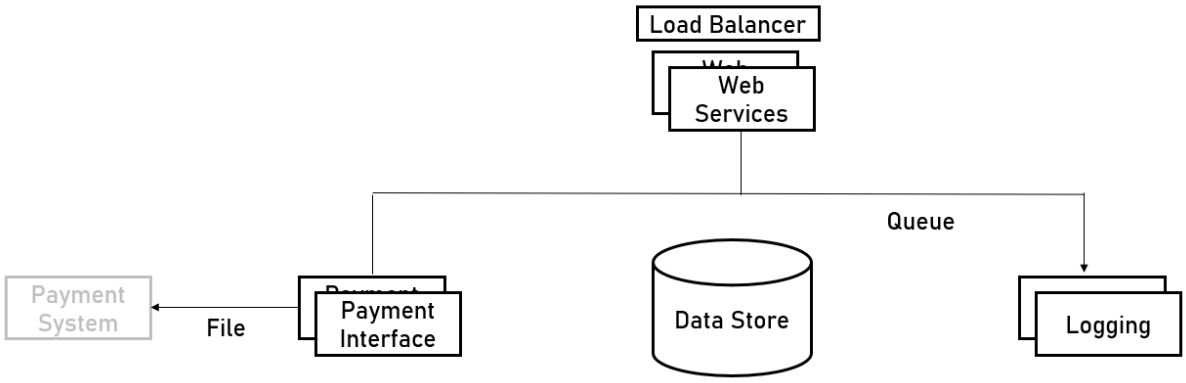
Logical Diagram



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.

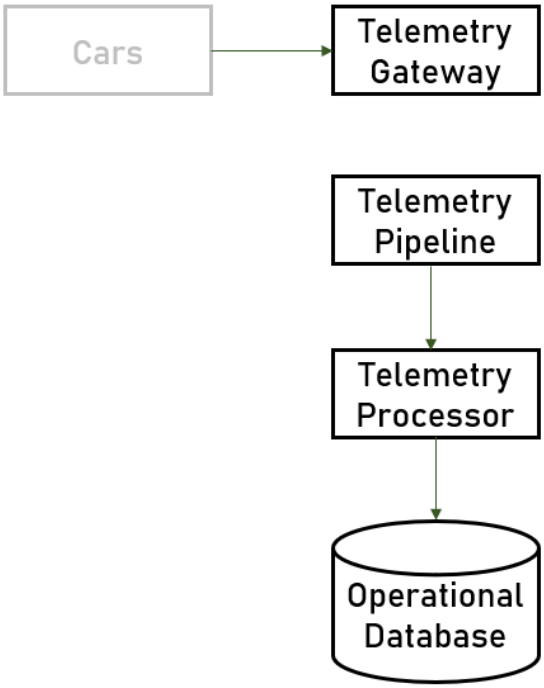
First, we need to put the *Cars* module on our architecture. Even though we do not design that specific software, it is the source of our system's data.

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.

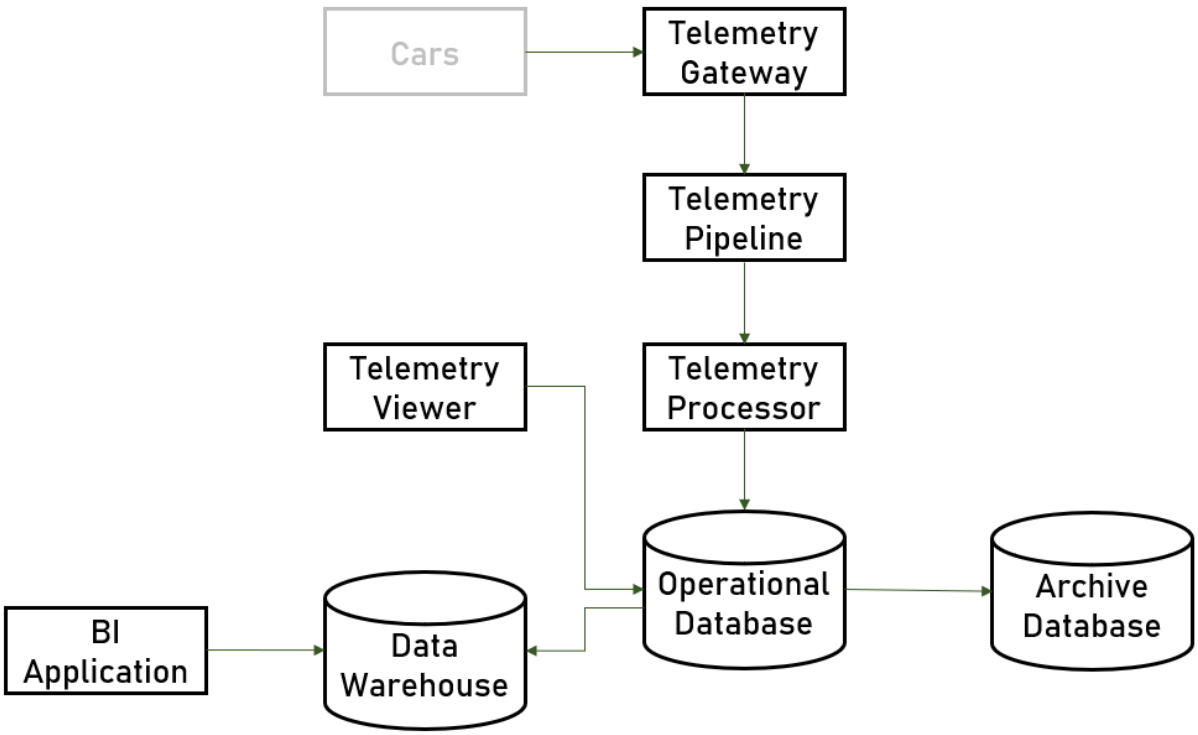
Next to the operational database, we will introduce a *Data Warehouse* database, responsible for data aggregation. It is always a good idea to store the real-time data and the reports-related one in separate databases. If we don't do it, our business intelligence analysis may hinder the real-time dashboards'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



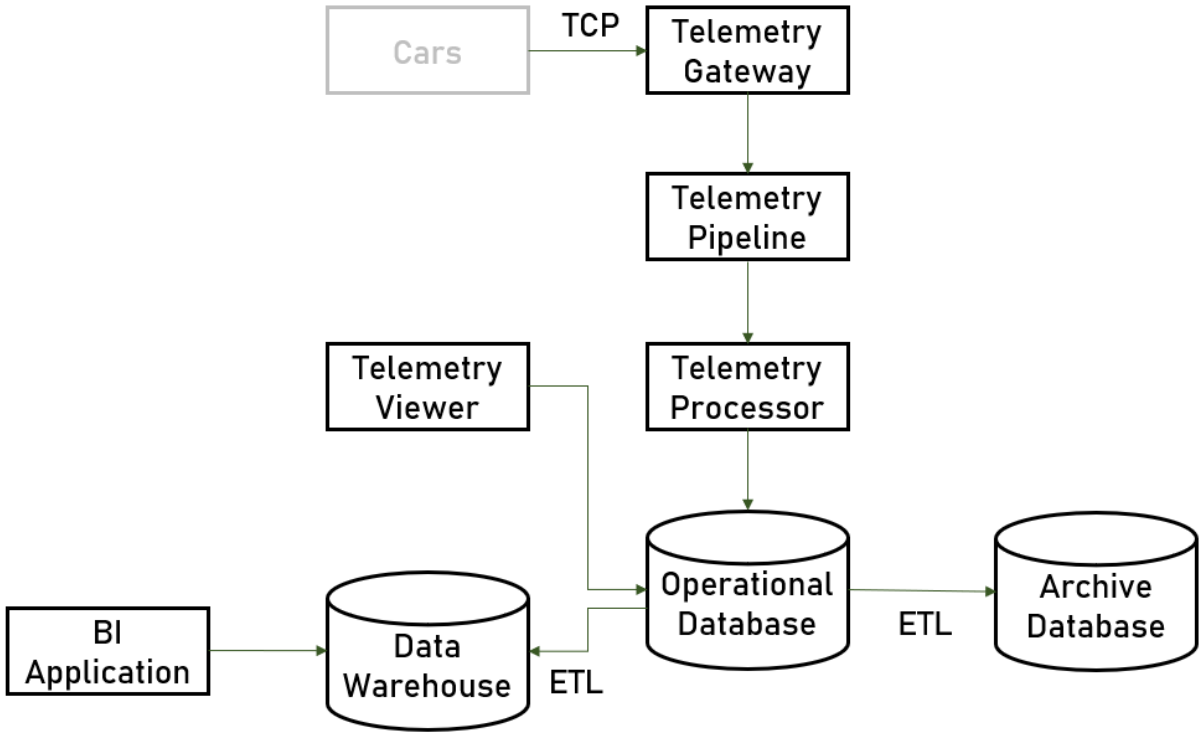
It's time for messaging. We need to discuss how these modules will communicate with each other, and we need to decide the transport mechanisms behind each service's contract.

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."

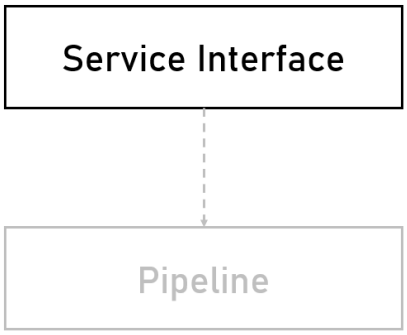
The primary server-side expertise of the company's developers is *Python*. However, we cannot use *Python* for this module, as it is relatively slow. We are expecting a huge load and must provide outstanding performance, which *Python* simply cannot deliver. It is interpreted language, and it doesn't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

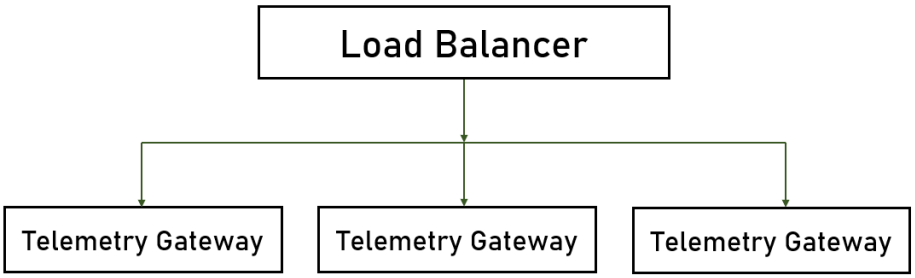
What about redundancy? How would you handle it?

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*.

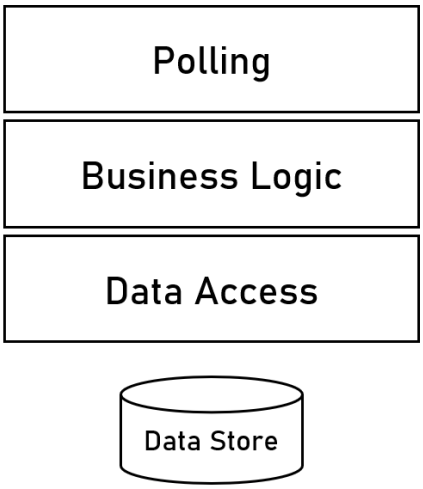
Great, we've chosen the processor and the operational database, but there is another one – the archive. What are the main requirements here? We need excellent support for a considerable amount of data,

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*.

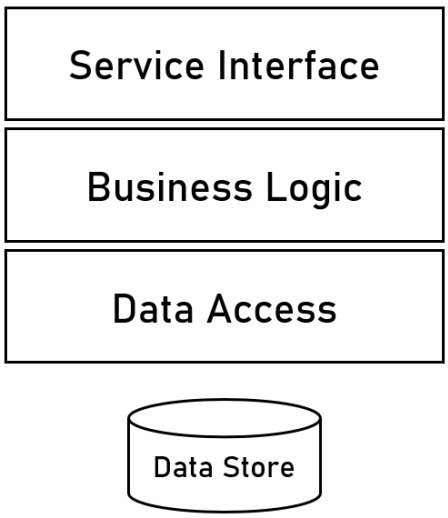
You will not receive any detailed instructions here, so try to independently create the whole module architecture without the book's help.

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.

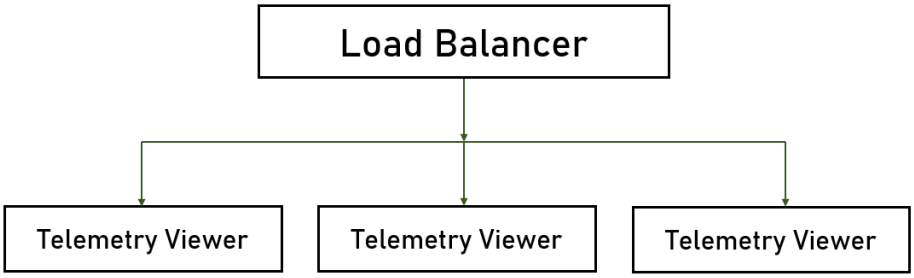
Here is the API specification:

Functionality	Path	Return Codes
Get latest errors for all cars	`GET /api/v1/telemetry/errors`	`200 OK`
Get latest telemetry for specific car	`GET /api/v1/telemetry/{carId}`	`200 OK` `404 Not Found`
Get latest errors for specific car	`GET /api/v1/telemetry/errors/{carId}`	`200 Ok` `404 Not Found`

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.

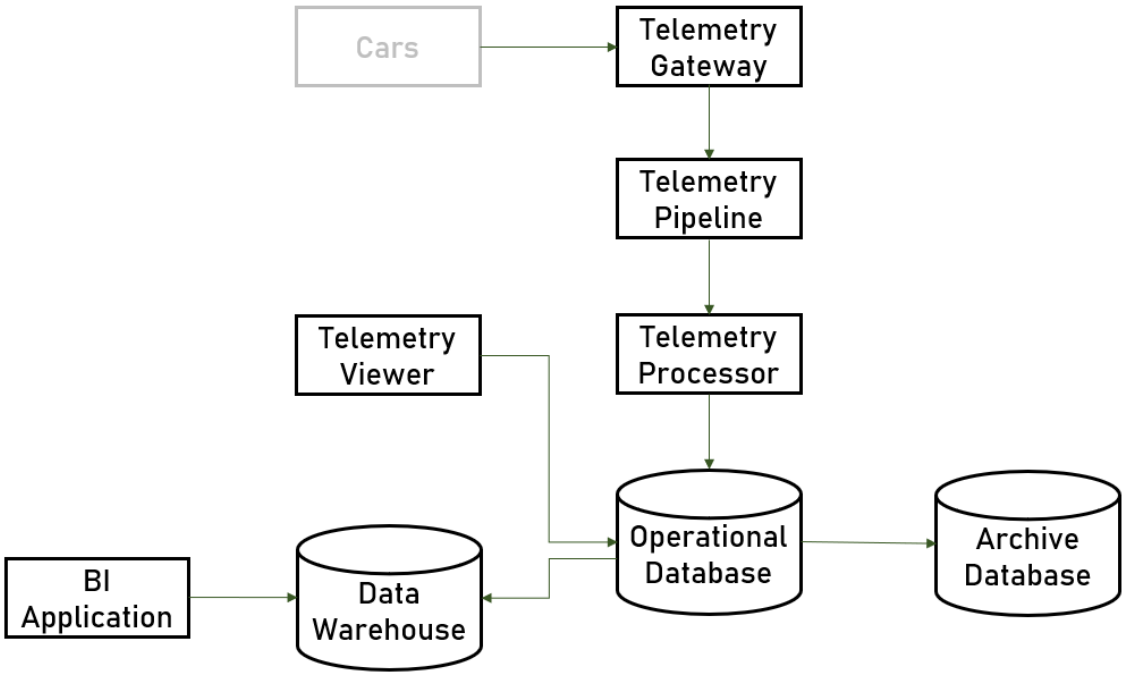
What should we do in terms of infrastructure?

Are we going to include container and orchestration in the system?

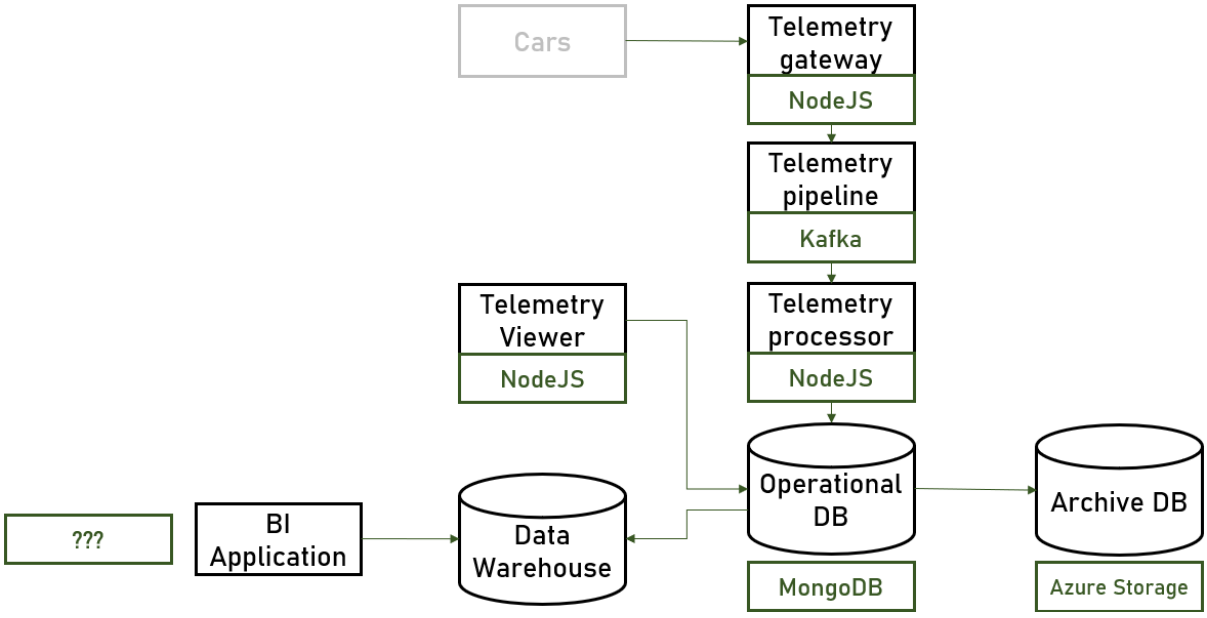
Do we need a CI/CD pipeline?

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

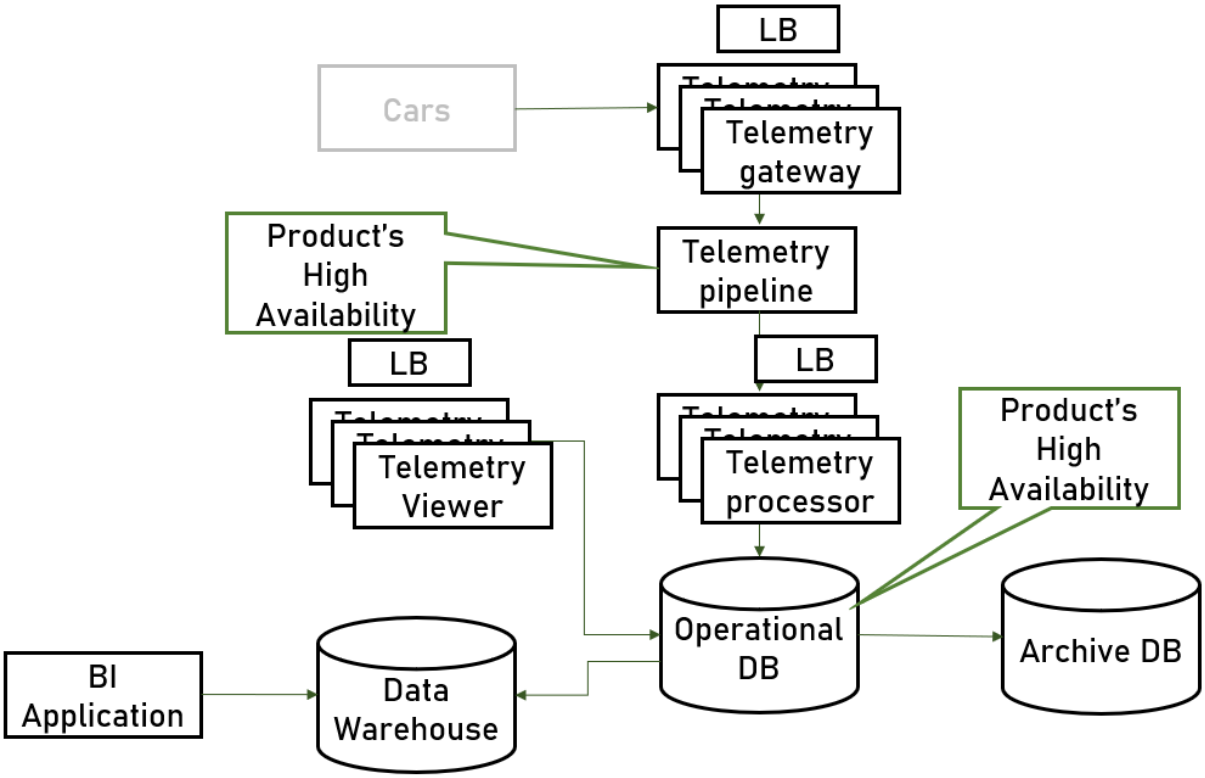
Logical Diagram



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.

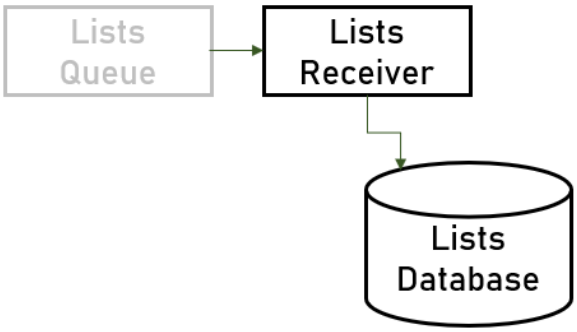
After you are ready, you can continue reading to find one of the possible solutions.

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.

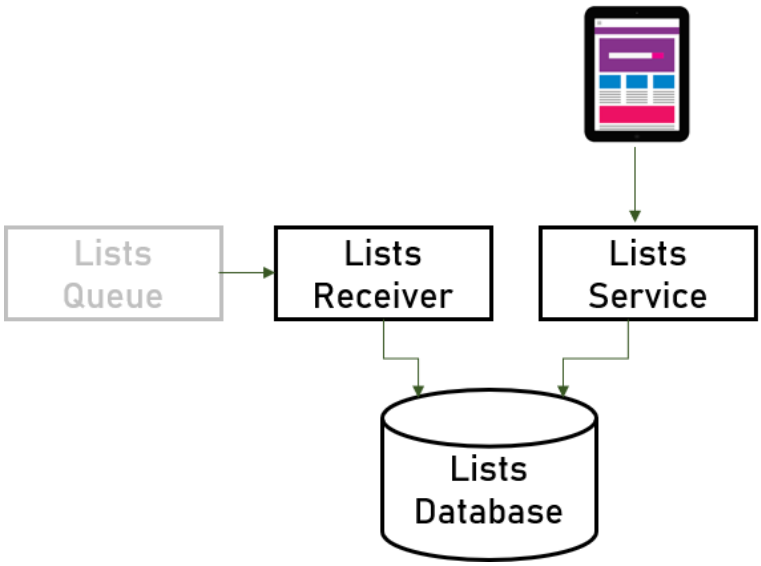
The next module is the one that receives the lists from the queue. Let's name it *List Receiver*. This component is also responsible for storing the list records in a database:



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*.

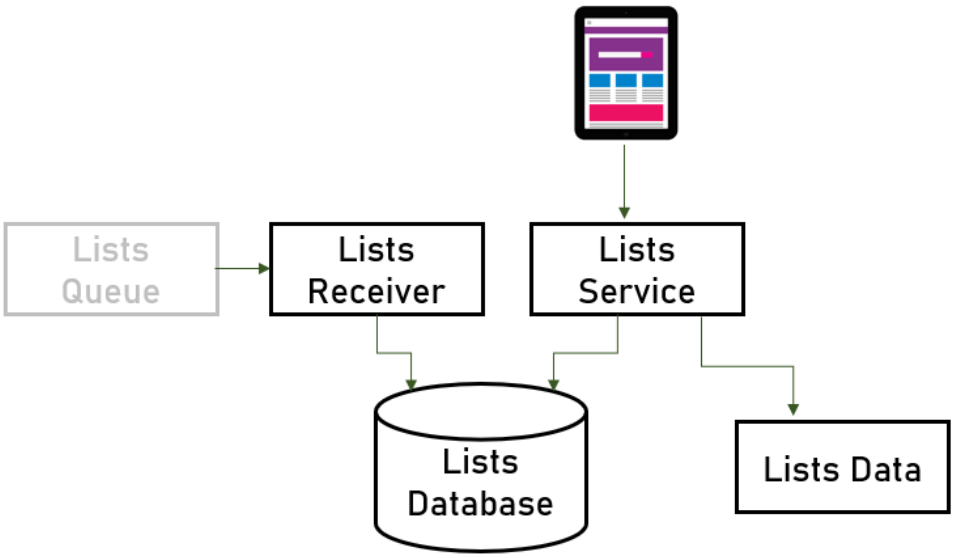
But why do we separate the receiver and the handler? They could work together in a single module. The answer is simple yet important - because we want the service to operate independently of the data source. By separating the responsibility between two modules, we successfully achieve loose coupling. 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:



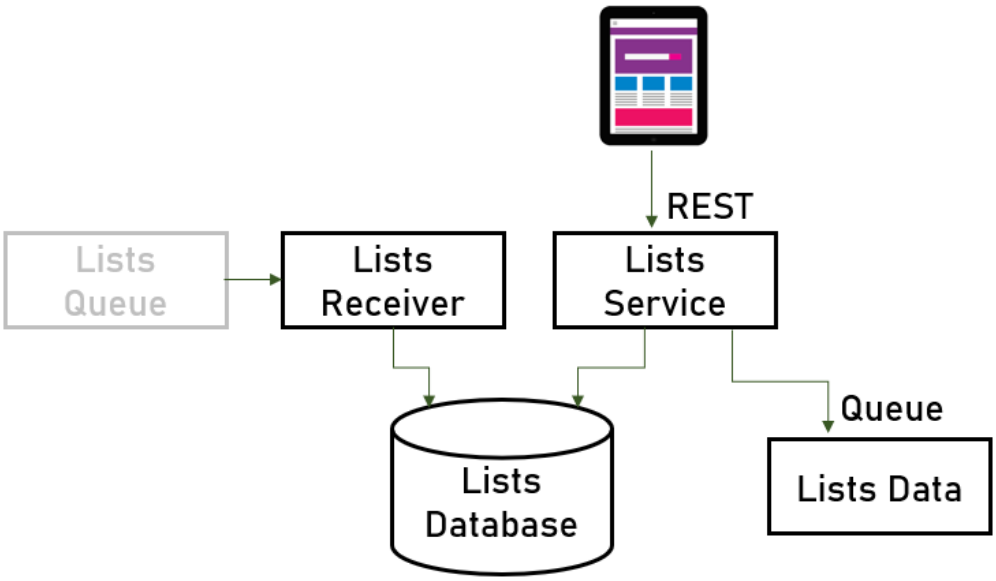
It's time for messaging. Once again, we need to discuss how these modules will communicate with each other.

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.

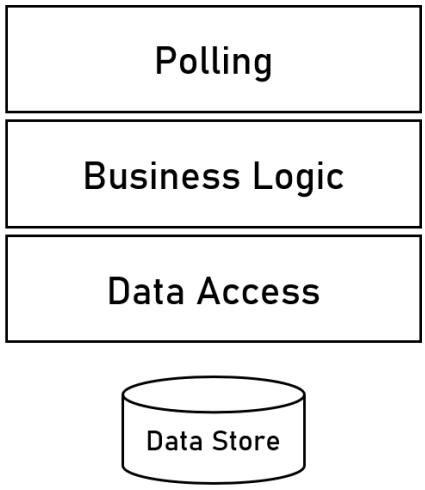
Partitioning allows us to separate the data into groups based on one or more columns. In our case, we can partition the records by their month of creation. We will have a partition for January, for February, and so on.

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?

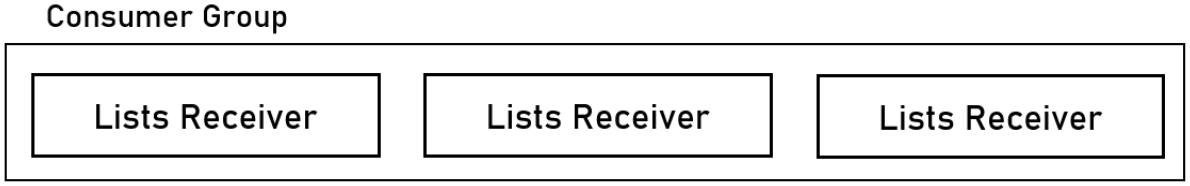
Of course, you can. You are quite a machine after practicing with this book. But for the sake of completeness, here it is:



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*:

Functionality	Path	Return Codes
Get next list to be processed	`GET /api/v1/lists/next?location=…`	`200 OK` `400 Bad Request`
Mark item as collected / unavailable	`PUT /api/v1/list/{listId}/item/{itemId}`	`200 OK` `404 Not Found`
Export list's payment data	`POST /api/v1/list/{listId}/export`	`200 Ok` `404 Not Found`

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.

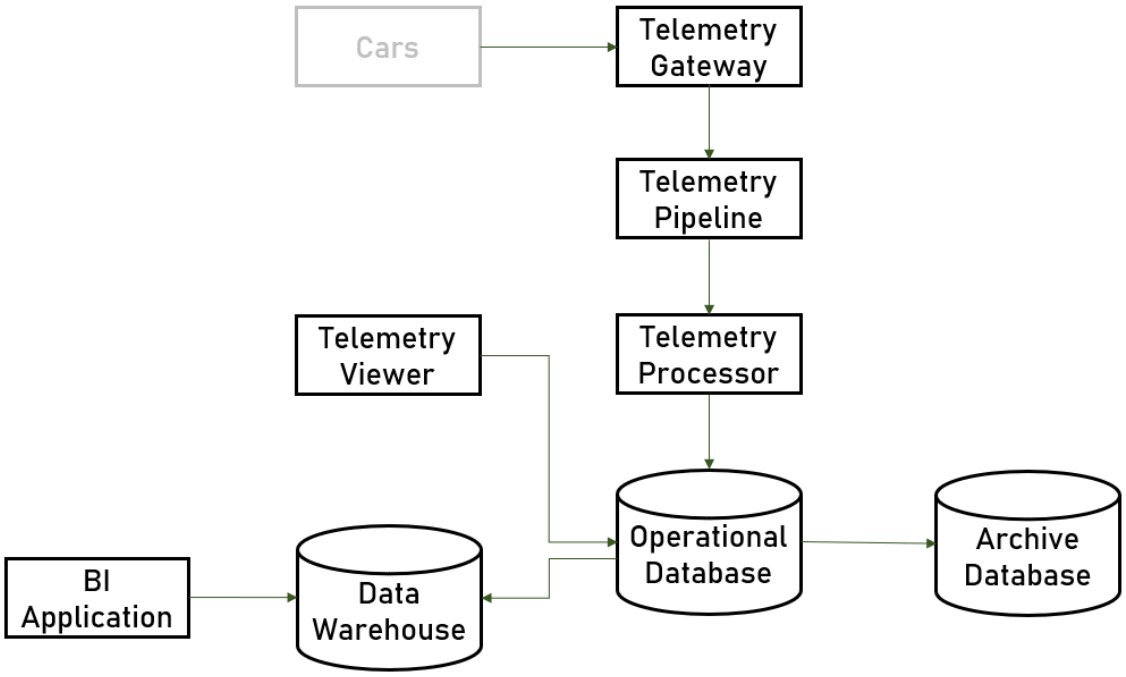
What should we do in terms of infrastructure?

Are we going to include container and orchestration in the system?

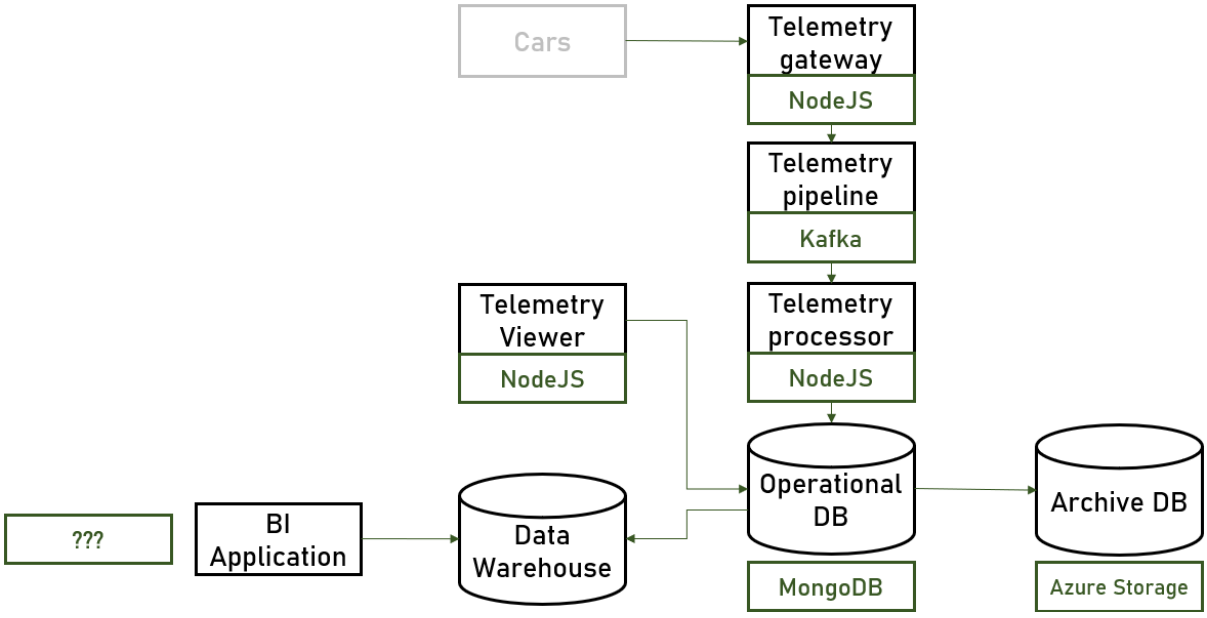
Do we need a CI/CD pipeline?

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

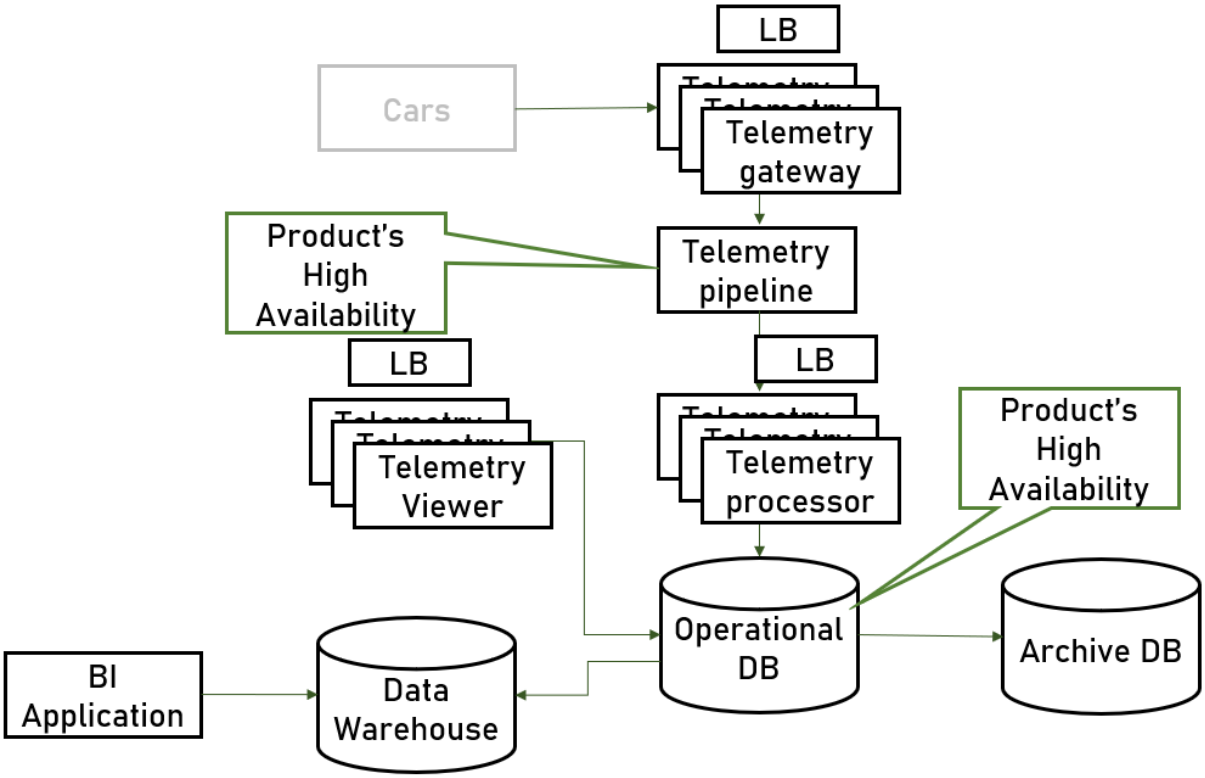
Logical Diagram



Technical Diagram



Physical Diagram



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