50 Popular Backend Developer Interview Questions and Answers

roadmap.sh/questions/backend

Explain what an API endpoint is?

An API endpoint is a specific URL that acts as an entry point into a specific service or a functionality within a service.

Through an API endpoint, client applications can interact with the server sending requests (sometimes even with data in the form of payload) and receive a response from it.

Usually, each endpoint can be mapped to a single feature inside the server.

Can you explain the difference between SQL and NoSQL databases?

SQL databases (or relational databases as they're also known) rely on a predefined schema (or structure) for their data. Whenever you describe a record, or table inside the database, you do so through its format (name and fields).

In NoSQL databases, there is no schema, so there is no predefined structure to the data. You usually have collections of records that are not obligated to have the same structure, even if they represent conceptually the same thing.

What is a RESTful API, and what are its core principles?

For an API to be RESTful (which means it complies with the REST guidelines), it needs to:

- It needs to follow a client-server architecture (which all HTTP-based services do).

- It has to provide a uniform interface which means:
    - There should be a way to identify resources from each other through URIs (Unique Resource Identification).
    - There should be a way to modify resources through their representation.
    - Messages should be self descriptive, meaning that each message should provide enough information to understand how to process it.
    - Clients using the API should be able to discover actions available for the current resource using the provided response from the server (this is known as HATEOAS or Hypermedia as the Engine of Application State).
- It needs to be stateless, which means each request to the server must contain all information to process the request.
- It should be a layered system, meaning that client and server don't have to be connected directly to each other, there might be intermediaries, but that should not affect the communication between client and server.
- Resources should be cacheable either by client or by server.
- Optionally, the server could send code to the client for it to execute (known as "Code on Demand").

Can you describe a typical HTTP request/response cycle?

The HTTP protocol is very structured and consists of a very well-defined set of steps:

- **Open the connection.** The client opens a TCP connection to the server. The port will be port 80 for HTTP connections and 443 for HTTPS (secured) connections.
- **Send the request.** The client will now send the HTTP request to the server. The request contains the following information:
    - An HTTP method. It can be any of them (i.e. GET, POST, PUT, DELETE, etc).
    - A URI (or Uniform Resource Identifier). This specifies the location of the resources on the server.
    - The HTTP version (usually HTTP/1.1 or HTTP/2).
    - A set of headers. They include extra data related to the request; there is a full list of HTTP headers that can be used here.
    - The optional body. Depending on the type of request, you'll want to also send data, and the data is encoded inside the body of the request.
- **Request processed by the server.** At this stage, the server will process the request and prepare a response.

- **Send the HTTP response back to the client.** Through the open channel, the server sends back an HTTP response. The response will contain the following elements:
    - The HTTP Version.
    - The status code. There is a list of potential status codes that describe the result of the request.
    - A set of headers with extra data.
    - The optional body, just like with the request, the body of the response is optional.
- **The connection is closed.** This is usually the last step, although with newer versions of the protocol, there are options to leave the channel open and continue sending requests and responses back and forth.

How would you handle file uploads in a web application?

From a backend developer perspective, the following considerations should be taken into account when handling file uploads regardless of the programming language you're using:

- **Perform server-side validations.** Validate that the size of your file is within range, and that the file is of the required type. You can check this OWASP guide for more details.
- **Use secure channels.** Make sure the file upload is done through an HTTPS connection.
- **Avoid name collision.** Rename the file ensuring the new filename is unique within your system. Otherwise this can lead to application errors by not being able to save the uploaded files.
- **Keep metadata about your files.** Store it in your database or somewhere else, but make sure to keep track of it, so you can provide extra information to your users. Also, if you're renaming the files for security and to avoid name collisions, keep track of the original filename in case the file needs to be downloaded back by the user.

What kind of tests would you write for a new API endpoint?

As backend developers, we have to think about the types of tests that there are out there.

- **Unit tests:** Always remember to only test the relevant logic through the public interface of your code (avoid testing private methods or inaccessible functions inside your modules). Focus on the business logic and don't try to test the code that uses external services (like a database, the disk or anything outside of the piece of code you're testing).
- **Integration tests:** Test the full endpoint through its public interface (the actual HTTP endpoint) and see how it integrates with the external services it's using (i.e the database, another API, etc).

- **Load testing / performance testing:** You might want to also check how the new endpoint behaves under heavy stress. This might not be required depending on the API you're using, but as a rule of thumb, it's a good one to perform at the end of the development phase before releasing the new endpoint into prod.

Describe how session management works in web applications

The following is a high-level overview of how session management works for web applications:

- **The session is created.** This happens with the first interaction with the system by the user (during log-in). The backend of your app will create a unique session ID that will be stored and returned to the user to use in future requests.
- **Session information storage.** The session data needs to be stored somewhere. Whether it's in-memory, or inside a database, it needs to be indexed by the session ID from the previous point. Here the best option is to use a database (ideally something like Redis with high I/O performance) so that the services can be scaled independently from the session data.
- **The session ID is sent to the client.** The most common way of doing this is through cookies. The backend can set up a cookie with the session ID and the frontend can read it securely and use that ID however it needs to.
- **Client sends the session ID.** After the ID is created, the client application will identify itself with the backend using this ID on every request.
- **Accessing the session data in the backend.** The backend will access the stored session data using the session ID received from the client.
- **Session is closed.** After a while, or perhaps through a user action, the session ID will be deleted, which will cause the session data to be lost (or removed from the DB). This effectively ends the interactions between the client and the server as part of the existing session.

How do you approach API versioning in your projects?

There are several ways in which you can handle API versioning, but the most common ones are:

- **Keeping the version in the URL:** Either as a URL attribute (i.e /your-api/users?v=1) or as part of the URL (i.e /v1/your-api/users). In both situations the version is clearly visible to the developer using the API.
- **Using a custom header:** Another option is to have a custom header (such as api-version) where the developer must specify the version of your API they intend to use.

How do you protect a server from SQL injection attacks?

There are many ways to protect your relational database from SQL injection attacks, but here are three very common ones.

- **Prepared statements with parameterized queries.** This is probably the most effective way since it's done by a library or framework, and all you have to do is write your queries leaving placeholders for where the data is meant to go, and then, in a separate place, provide the actual data.
- **Use an ORM (Object-Relational Mapping).** These frameworks allow you to abstract the interaction with your database and create the SQL queries for you, taking into account all matters of security around that interaction.
- **Escaping data.** If you want to do this manually, you can take care of escaping special characters that might break how you construct your SQL queries. Keeping a list of blacklisted characters to escape in this situation is a good idea, so you can programmatically go through them.

Explain the concept of statelessness in HTTP and how it impacts backend services

HTTP is, by design, a stateless protocol, which means that every request is unique and unrelated to any previous request, even from the same client.

This affects backend web services by forcing them to implement their own state management solutions if such a feature is required.

What is containerization, and how does it benefit backend development?

It's a lightweight virtualization method to package applications and their dependencies, ensuring consistent environments across different systems.

It's actually a benefit for backend development because it provides isolation and portability by simplifying deployment processes and reducing conflicts between software versions and configurations.

What measures would you take to secure a newly developed API?

There are many ways to secure an API, here are some of the most common ones:

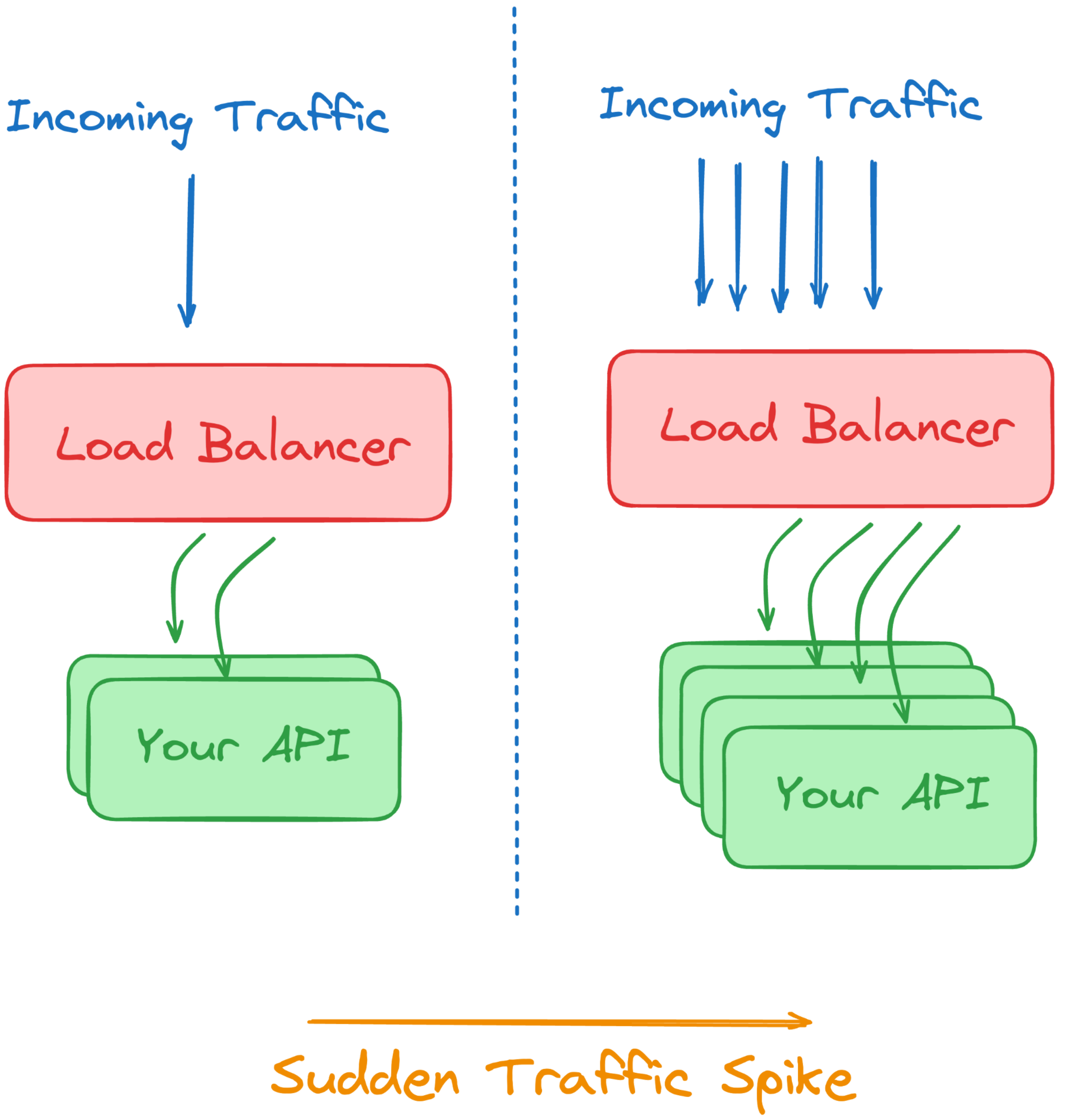
- Add an authentication method, such as OAuth, JWT, Bearer tokens, Session-based auth, and others.
- Use HTTPS to encrypt data transfer between client and server.

- Configure strong CORS policies to avoid unwanted requests.
- Setup a strong authorization logic, to ensure clients only access resources they have access to.

How would you scale a backend application during a traffic surge?

The most common way to scale up a backend application during traffic surges is to have multiple instances of the application behind a load balancer, and when the traffic surge happens, simply add more instances of the application.

This is known as horizontal scaling and works best when the backend application is stateless.



What tools and techniques do you use for debugging a backend application?

If the backend application being debugged is in the local dev machine, a simple solution would be to use the IDE itself. Most modern IDEs, such as IntelliJ, Eclipse and others have integrated debugging capabilities.

If the backend application is on the server though, you'll have to use other techniques, such as logging, which you can do with logging libraries. Or, you can use more complex tools such as JProfiler or NewRelic.

How do you ensure your backend code is maintainable and easy to understand?

The trick here is to follow best practices and coding standards such as:

- Modularity.
- Following naming conventions.
- Adding code comments.
- Doing regular refactors to keep technical debt under check.
- Keeping error handling messages consistent throughout the platform.
- Performing unit tests on all written code.

Describe how you would implement a full-text search in a database.

You can use the native full-text search functionality of a database, such as MySQL, Postgre or even ElasticSearch.

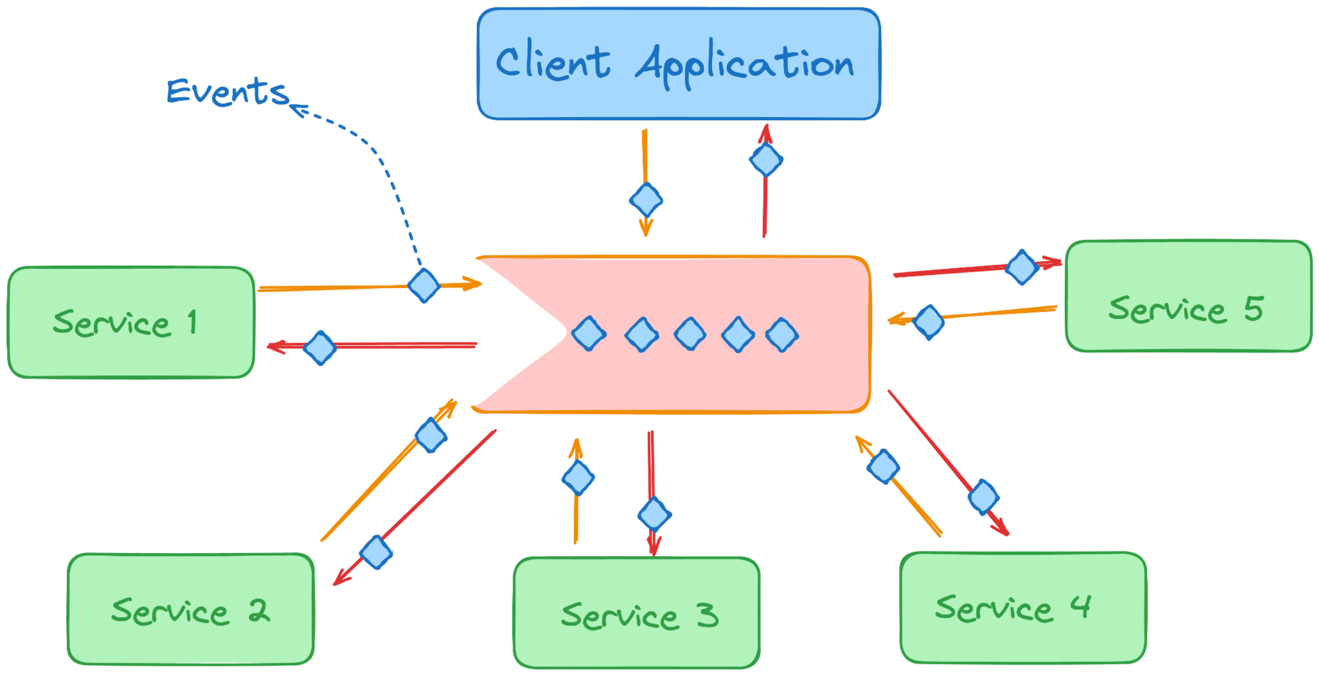
However, if you want to implement it yourself, the steps would be:

- Preprocessing the text data to be searched and normalizing it by applying tokenization, stemming and removing stop words.
- Then, implement an inverted index, somehow relating each unique word to the records that contain that word.
- Create a search UI and normalize the input from the user in the same way the text data was normalized.
- Then, search for each word in the database.
- Sort the results by implementing a scoring logic based on different aspects, such as word frequency.

How would you approach batch processing in a data-heavy backend application?

The best option here would be to use a batch-processing framework such as Hadoop or Spark. They are already prepared to process massive amounts of data in parallel.

Can you explain the use and benefits of a message queue in a distributed system?



A message queue in a distributed system can act as the core component of a reactive architecture. Each service can trigger and listen for events coming from the queue. That way, when the events arrive, those services can react to them without having to actively poll other services for a response.

What strategies would you use to manage database connections in a high-load scenario?

During a high-load scenario, there are several things a developer can do to improve the performance of the database connection:

- Using connection pools to reuse connections reduces the time required to establish a new one.
- Load balancing the database traffic (the queries) between a group of databases would help distribute the load.
- Even optimizing your queries can reduce the time you're using each connection, helping you optimize the use of resources and minimizing the time you're spending with each active connection.

How would you set up a continuous integration/continuous deployment (CI/CD) pipeline for backend services?

There are multiple considerations to have while setting up Continuous Integration and Continuous Delivery pipelines:

- **Using source control** as the trigger for the entire process (git for example). The build pipelines for your backend services should get executed when you push your code into a specific branch.
- **Pick the right CI/CD** platform for your needs, there are many out there such as GitHub Actions, GitLab CI/CD, CircleCI and more.
- Make sure you have **automated unit tests** that can be executed inside these pipelines.
- **Automatic deployment** should happen only if all tests are executed successfully, otherwise, the pipeline should fail, preventing broken code from reaching any environment.
- **Use an artifact repository** such as JFrog Artifactory or Nexus Repository to store successfully built services.
- Finally, consider setting up a **rollback strategy** in case something goes wrong and the final deployed version of your service is corrupted somehow.

Can you describe a distributed caching strategy for a high-availability application?

In this scenario, you have to consider the following points:

- Implement a **cluster of servers** that will all act as the distributed cache. Implement a **data sharding** process to evenly distribute the data amongst all cache servers and make sure it uses a consistent hashing algorithm to minimize cache reorganization when a server joins or leaves the cluster.
- Add **cache replication** to have redundancy of your data in case of a failure, that way, your distributed cache is fault-tolerant as well.
- **Cache invalidation** is a must on any caching solution, as your data will become stale if you don't update it often.

What methods can you use for managing background tasks in your applications?

It highly depends on your tech stack and what those background tasks are doing. And because of that, there are many options:

- Using task queues such as RabbitMQ or Amazon SQS. These will let you have workers in the background as secondary processes while your application continues working.
- There are background job frameworks such as Celery for Python or Sidekiq for Ruby.
- You can also just rely on cron jobs if you want.
- If your programming language permits it, you can also use threads or workers to run these tasks in the background but within the same application.

How do you handle data encryption and decryption in a privacy-focused application?

For this type of application, you have to distinguish between “data at rest” and “data in transit”. The first one describes your data while it’s stored in your database (or any data storage you have). And the latter (data in transit) describes your data while it’s traveling between backend services or even between the server and the client.

For “data in transit”, you should be ensuring that connection happens inside a secure and encrypted channel such as HTTPS.

And for “data at rest” use strong encryption algorithms such as AES, RSA or ECC and make sure to keep their associated keys somewhere safe, such as inside a dedicated secrets management tool or key management services (KMS).

What are webhooks and how have you implemented them in past projects?

Webhooks are user-defined HTTP callbacks, they are triggered by a specific event inside a system. They’re mainly used to notify about results of multi-step, asynchronous tasks to avoid keeping an open HTTP connection.

As for the implementation of a webhook, consider the following:

- **Event definition.** Make sure to define exactly what events will trigger the message to the webhook and the type of payload associated with those events.
- **Endpoint creation.** Based on the previous step, define an HTTP endpoint that can deal with the expected request (especially with the payload part). In other words, if you’re receiving data in the webhook request, make sure to create the endpoint as a POST endpoint, otherwise you can use a GET one.
- **Security.** Remember to implement some form of security measures around your webhook endpoint so it can’t be exploited.

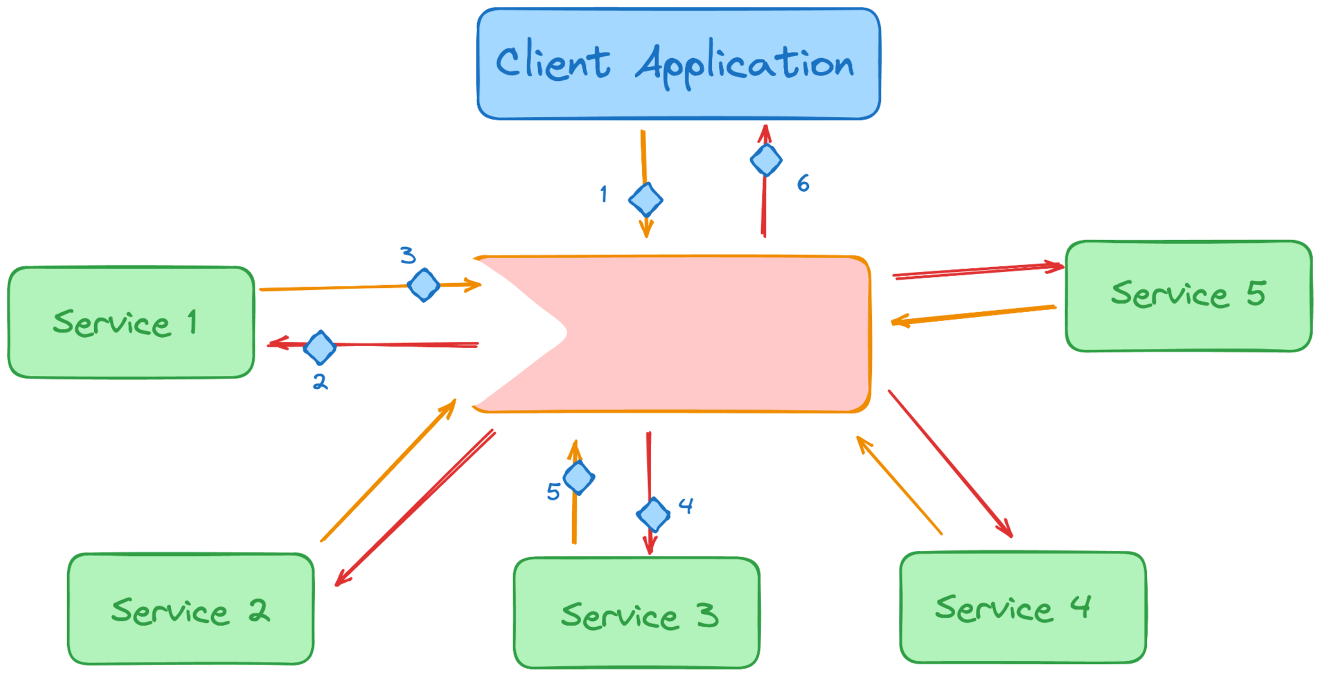
What considerations must be taken into account for GDPR compliance in a backend system?

The following are key considerations to be taken into account:

- Capture only what you need and what you told your users you’d capture. Remember that to comply with GDPR, you have to ask for your user’s consent to collect their data, and you have to specify the actual data points you’re collecting. So focus on those and nothing else.
- Secure your data. As part of the regulations, you have to make sure your data is secured both in transit and at rest. There are regular security audits that have to happen to ensure security is kept high.

- The user has rights over the data you've captured, so make sure you give them the right endpoints or services to read it, edit it or even remove it if they want.

Explain how you would deal with long-running processes in web requests.



For web requests that trigger long-running processes, the best option is to implement a reactive architecture. This means that when a service receives a request, it will transform it into a message inside a message queue, and the long-running process will pick it up whenever it's ready to do so.

In the meantime, the client sending this request receives an immediate response acknowledging that the request is being processed. The client itself can also be connected to the message queue (or through a proxy) and waiting for a "ready" event with its payload inside.

Discuss the implementation of rate limiting to protect APIs from abuse.

To implement rate limiting, you have to keep the following points in mind:

- **Define your limits.** Define exactly the amount of requests a client can make. This can be measured in requests per minute, per day, or per second.
- **Choose a limiting strategy.** Pick a rate-limiting algorithm, like the fixed window counter, sliding log window, token bucket, or leaky bucket. You can read more about these algorithms here.

- **Store your counters somewhere.** Use a fast data store (like Redis) to keep track of the number of requests or timestamps for each client.
- Once the limit is reached, try to respond with a **standard status code**, such as 429 which indicates there have been “Too Many Requests”.

If you want to take this further, you can look into using an existing API Gateway that already provides this functionality or look into adding support for sudden bursts of traffic to avoid penalizing clients that are slightly above the limits every once in a while.

How do you instrument and monitor the performance of backend applications?

A great way to monitor the performance of backend applications is to use an Application Performance Management system (APM) such as New Relic, AppDynamics or even Dynatrace.

Those will track your application’s performance and provide insight into the bottlenecks you might have with minimum effort on your part.

What are microservices, and how would you decompose a monolith into microservices?

Microservices are a software architecture style that allows you to structure your backend applications as a collection of independent services, each one working around a specific business need.

If you’re looking to decompose a monolith into a set of microservices, you have to keep the following points in mind:

- Start by identifying the logical boundaries of your monolith. Its inner logic will tackle multiple responsibilities and types of resources. Find the boundaries between them to understand where one service starts and another one ends.
- Define your services based on the boundaries from the previous point and start decoupling the data needs as well. Either into multiple tables or even individual databases whenever it makes sense.
- Start incrementally refactoring the monolith and extracting the logic required for each individual microservice into its own project.

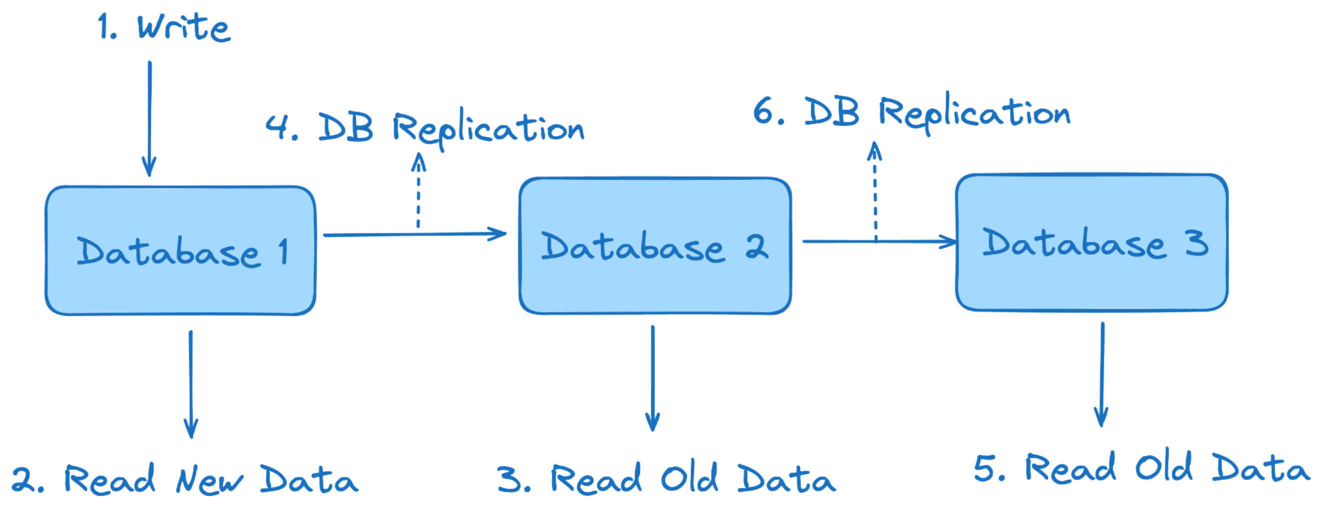
By the time you’re done, your original monolith should not be needed anymore, and all your microservices will have their own independent deployment pipeline and code repository.

How have you managed API dependencies in backend systems?

A great way to handle API dependencies in backend systems is to take advantage of API versioning. Through this simple practice, you can ensure that your systems are actually using the right API, even if there are multiple versions of it.

This also allows you to have multiple backend systems using different versions of the same API without any risk of inconsistency or of updates breaking your systems.

Describe the concept of eventual consistency and its implications in backend systems.



Eventual consistency is a consistency model used in distributed computing. This model guarantees that any piece of information written into a distributed system will become consistent (meaning that all servers will have the same version of this data) eventually as opposed to others where immediate consistency is assured.

For backend systems this implies that there is a need for data synchronization between all parts of the distributed system and on top of that, a potential need to resolve data conflicts, if different parts of your system are dealing with different versions of the same data record.

What is a reverse proxy, and how is it useful in backend development?

A reverse proxy is a server that sits in front of multiple other servers and redirects traffic to those web servers based on different logic rules. For example, you could have two web servers, one for customers of your business and another one for your employees.

You could configure a reverse proxy to redirect traffic to one or the other depending on the value of a header sent in the request or the actual URL being requested.

It is very useful in backend development because it allows you to do many different things, for example:

- Load balancing traffic between multiple instances of the same backend service.
- Provide an extra layer of security by hiding the location of the backend services and handling attacks, such as DDoS.
- It can cache content, reducing server load on your web servers.
- It allows you to switch backend services without affecting the public-facing URLs.

How would you handle session state in a load-balanced application environment?

In a load-balanced application scenario, the main issue with session state is that if the backend system is handling session data in memory, then subsequent requests from the same client need to land on the same server, otherwise session data is fragmented and useless.

There are two main ways to solve this problem:

- Sticky sessions: This allows you to configure the load balancer to redirect requests from the same client into the same server every time. The downside with this one, is that the traffic is not always equally distributed among all copies of your backend services.
- Centralized session store: This solution involves taking the session data outside of your backend services into a centralized data store that all copies of your service can access. This makes it easier on the load balancer, but requires extra logic and more “moving parts”.

It’s up to you and your specific technical requirements to determine which strategy works best for you.

What is database replication, and how can it be used for fault tolerance?

Database replication implies the replication of data across multiple instances of the same database. In this scenario, there is usually one database that’s acting as a master to all clients that are connecting it, and the rest act as “slaves” where they simply receive updates on the data being changed/added.

The two main implications of this in fault tolerance are:

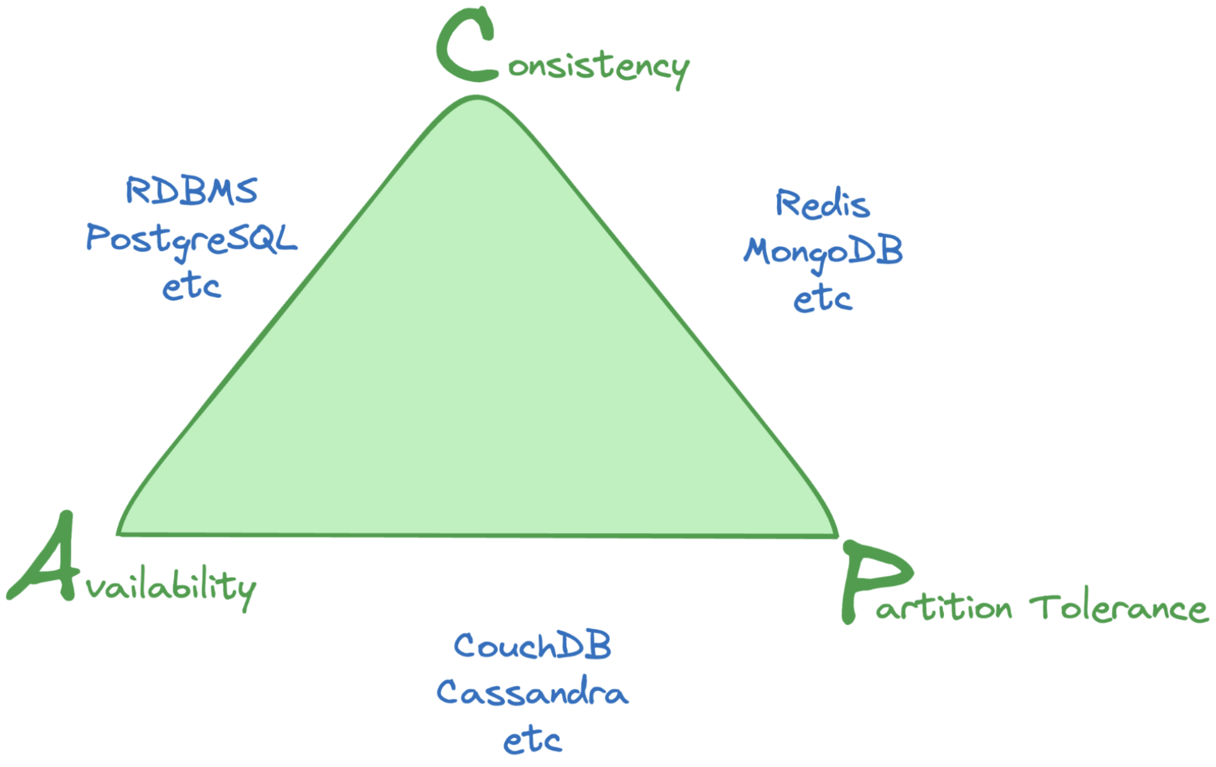
- A database cluster can withstand problems on the master server by promoting one of the slaves without losing any data in the process.

- Slaves can be used as read-only servers, increasing the amount of read requests that can be performed on the data without affecting the performance of the database.

Describe the use of blue-green deployment strategy in backend services

The blue-green strategy involves having two identical production environments, having one of them serving real traffic while the other is getting ready to be updated with the next release or just idle, waiting to be used as a backup.

Can you explain the consistency models in distributed databases (e.g., CAP theorem)?



The CAP theorem says that distributed databases can't simultaneously provide more than two of the following guarantees:

- Data Consistency: Meaning that every read is always returning the most recent result of the write operation. This is very relevant in this model because we're dealing with multiple servers and data needs to be replicated almost immediately to guarantee consistency.
- Availability: Meaning that every request will always receive a valid response.
- Partition tolerance: The distributed system continues to operate and work without data loss even during partial network outages.

For example, if the system is consistent and highly available, it won’t be able to withstand partial network outages. If on the other hand, the system is highly available and partition tolerant, it won’t be able to ensure immediate data consistency.

How do you manage schema migrations in a continuous delivery environment?

The two main aspects to consider when managing schema migrations, especially in CD environments are:

- Track your schema migrations inside version control. Keep these files versions with your code, as there is a direct relation between those versions.
- Use automated migration tools such as Flyway or Liquibase to simplify the process and keep it standard.

What strategies exist for handling idempotency in REST API design?

For REST APIs you can take advantage of HTTP verbs and define your idempotent operations using inherently idempotent verbs, such as GET, PUT, and DELETE.

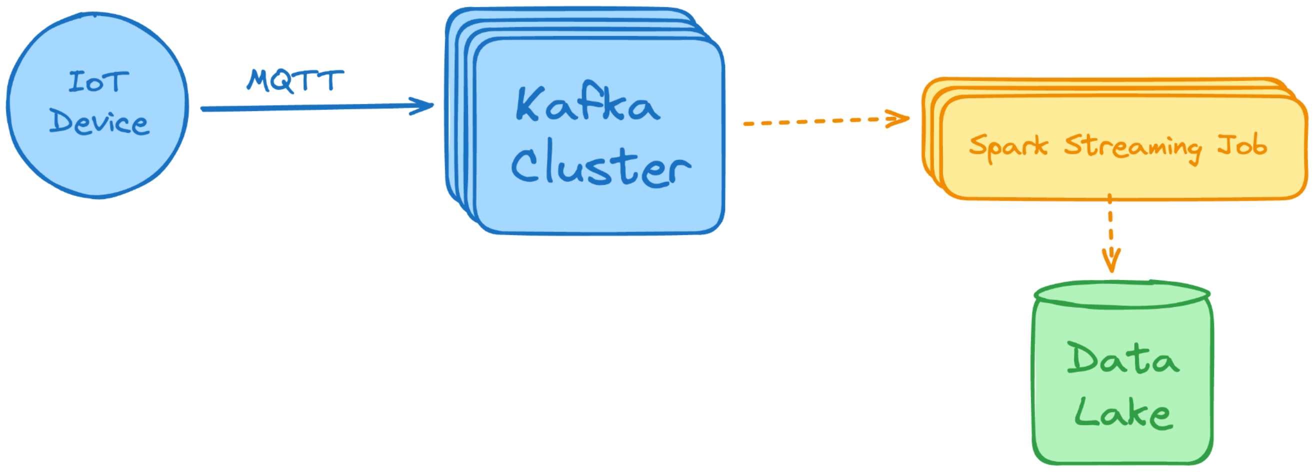
Or you can always manually implement a key-based logic to avoid repeating the same operation multiple times if the key provided by the client is always the same.

Describe the implementation of a single sign-on (SSO) solution

At a very high level, the steps required to implement an SSO solution are:

- Picking an identity provider, such as Okta or Keycloack.
- Each application will then integrate with the Identity provider from the previous step using a standard SSO protocol, such as SAML, OpenID or any other.
- For the first user access, the application will connect with the IdP and authenticate the user, getting an access token in return.
- Then during subsequent requests, the application will validate the provided token through the IdP.

Explain how you would develop a backend system for handling IOT device data streams



A real-time data capture and processing architecture would require the following components:

- Use a scalable data ingestion service such as Kafka or AWS Kinesis that is compatible with one of the many IoT standard protocols (like MQTT or CoAP).
- Process the data through real-time processing engines such as Apache Flink or Spark Streaming.
- Store the data inside a scalable data lake, ideally a time-series compatible system such as InfluxDB.

How would you architect a backend to support real-time data synchronization across devices?

If you want to support real-time data synchronization, you'll have to find a way to create stable and efficient communication channels between devices and find a way to solve potential data sync conflicts when several devices are trying to change the same record.

So, for communication channels, you can use one of the following:

- Socket-based bidirectional channels that allow for real-time data exchange.
- Using a pub/sub model to efficiently distribute data between multiple devices. You can use something like Redis or Kafka for this one.

For data conflict resolutions, you can use algorithms like Operational Transformation (OT) or Conflict-Free Replicated Data Types (CRDTs).

Discuss the benefits and drawbacks of microservice architectures in backend systems.

Benefits:

- **Scalability:** microservices can scale independently from each other.
- **Tech flexibility:** you can use different tech stacks depending on the particular needs of each microservice.
- **Faster deployments:** microservices can be deployed individually, improving the speed at which you deliver changes to production.

Drawbacks:

- **Over complex architecture.** In some situations, a microservice-based architecture can grow to be too complex to manage and orchestrate.
- **Debugging:** Debugging problems in a microservices-based architecture can be difficult as data flows through multiple services during a single request.
- **Communication overhead:** Compared to a monolithic approach, communication between microservices can be overly complex.

How would you approach load testing a backend API?

- First you have to understand the goals and set up a testing environment. Ideally, your environment would closely resemble production.
- Design and implement your tests with the tools you've selected (such as JMeter, LoadRunner or any other).
- Start to gradually increase the load on your tests while monitoring the performance and stability of your system.
- Optimize your backend API and go back to the first step to redesign your tests and try again until you're happy with the results.

Describe how you would implement a server-side cache eviction strategy.

To define this strategy, you'll need to define the following elements:

- The size limit that will trigger the cache eviction when exceeded.
- A monitoring strategy to determine if the eviction strategy is working properly or if it needs adjustment.
- A cache invalidation mechanism.
- And an eviction policy, which could be one of the following:
  - **LRU (Least Recently Used):** Evict the least recently accessed items.
  - **LFU (Least Frequently Used):** Remove items accessed least frequently.
  - **FIFO (First-In, First-Out):** Evict items in the order they were added.
  - **Random:** Randomly select items to evict.
  - **TTL (Time-To-Live):** Expire items after a certain time.

What are correlation IDs, and how can they be used for tracing requests across services?

Correlation IDs are unique identifiers added on requests done to distributed architectures to facilitate tracking of requests throughout the architecture. Remember that usually, when a request hits a distributed backend system, the data from the request passes through multiple web services before generating a response.

This makes it easy to understand the journey each request goes through to debug any potential problems or performance issues.

Explain the difference between optimistic and pessimistic locking and when to use each

Optimistic locking is a strategy that:

- Assumes conflicts are rare and don't happen that often.
- Allows for concurrent data access.
- Checks if there are conflicts before committing.
- It's best used in high-read, low-write scenarios.

Pessimistic locking, on the other hand, is a strategy that:

- Assumes conflicts to be very common.
- Locks data and prevents concurrent access.
- Holds these locks for the duration of a transaction.
- It's best suited for high-write scenarios or when data integrity is critical.

What methods would you use to prevent deadlocks in database transactions?

There are many ways to prevent deadlocks in DB transactions; some of the most common are:

- Using lock ordering to acquire locks in a consistent global order, avoiding circular wait conditions.
- Using timeouts for DB transactions to automatically kill long-running operations that could lead to deadlocks.
- Use of optimistic concurrency control where possible, to avoid holding locks for too long.

How would you secure inter-service communication in a microservices architecture?

Starting from the basis of understanding that your inter-service communication is meant to only happen inside private networks (ideally, no public traffic should reach these services), here are some recommendations:

- Use encrypted channels, such as TLS to prevent common attacks such as man-in-the-middle.
- Use an API gateway to manage and authenticate traffic that reaches this private network.
- Enforce authentication and authorization for inter-service messages, making sure that only valid microservices can reach each other, and when they do, they only have access to what it makes sense for them to have.

Discuss techniques for preventing and detecting data anomalies in large-scale systems

Some common techniques include:

- Adding and implementing validation rules to prevent invalid data entry. Through the definition of schemas and schema constraints to enforce certain minimum standards, you can prevent data anomalies from happening.
- Implement data versioning to easily revert back if there are any anomalies detected.
- Implement a strong data quality practice to ensure that whatever information enters your system is properly validated and flagged if required.

Describe the process of creating a global, high-availability data storage solution for a multinational application.

Building a highly available data storage involves multiple areas, including:

- **Multi-zone environments.** If you're going with cloud-based solutions (such as Azure, AWS, GCP or others) then you're likely to have this requirement met instantly (except for some specific regions in the world). This is to ensure availability even during partial network outages.
- **Data replication.** Ensure your data is being replicated between servers of all zones. This is to ensure that if there is a failure taking some servers down (or even entire zones) there is no data loss.
- **Load balancing.** Ensure the traffic is properly load-balanced between all your availability zones to ensure the lowest latency for all your clients.
- And then there are other requirements like setting up a proper data governance policy to ensure data access is regulated, as well as fully complying with your local data regulations (like GDPR).