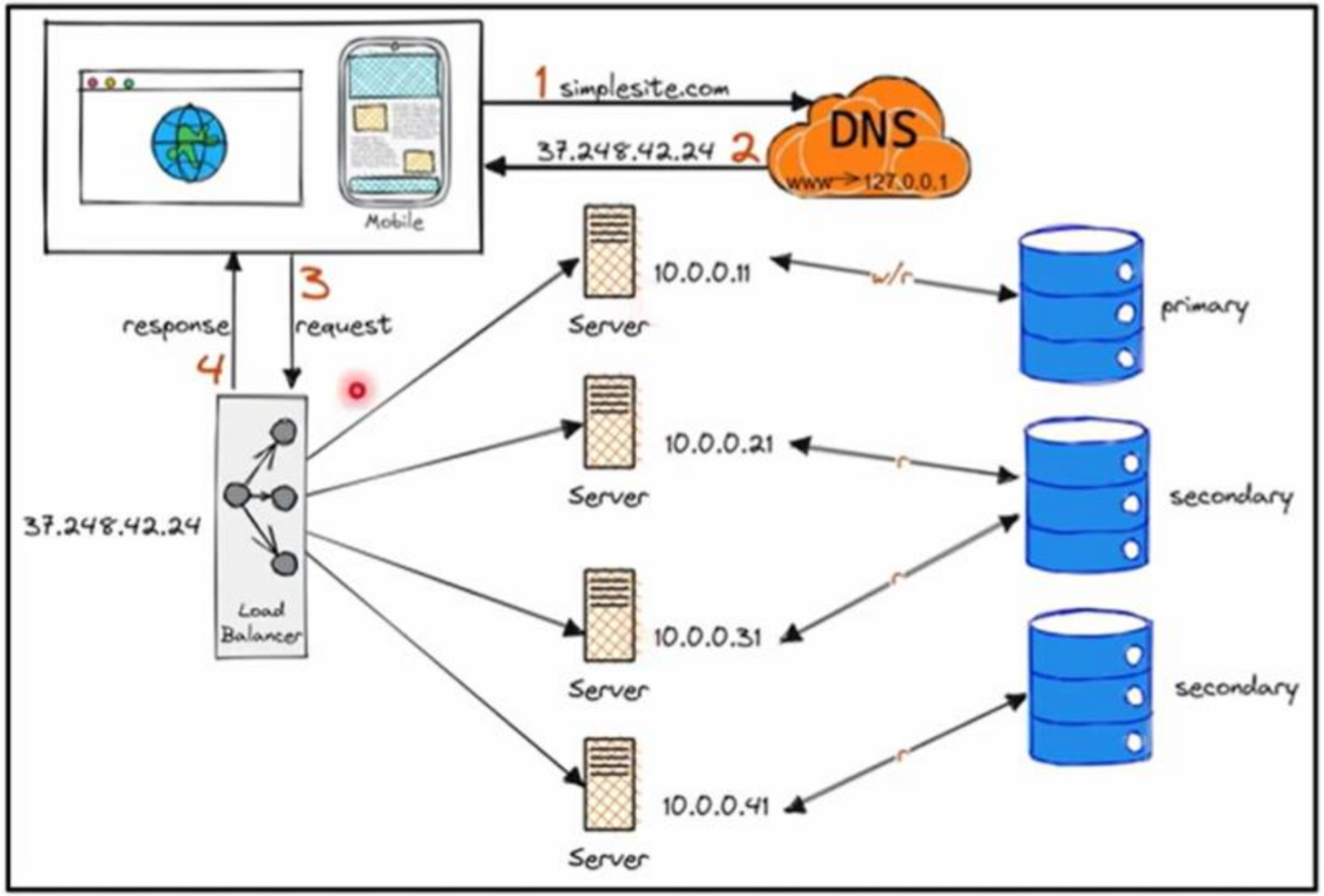
Software Design

Department of Computer Engineering
Sharif University of Technology
Maryam Ramezani
maryam.ramezani@sharif.edu

01

Stateless & Stateful

Save User Session

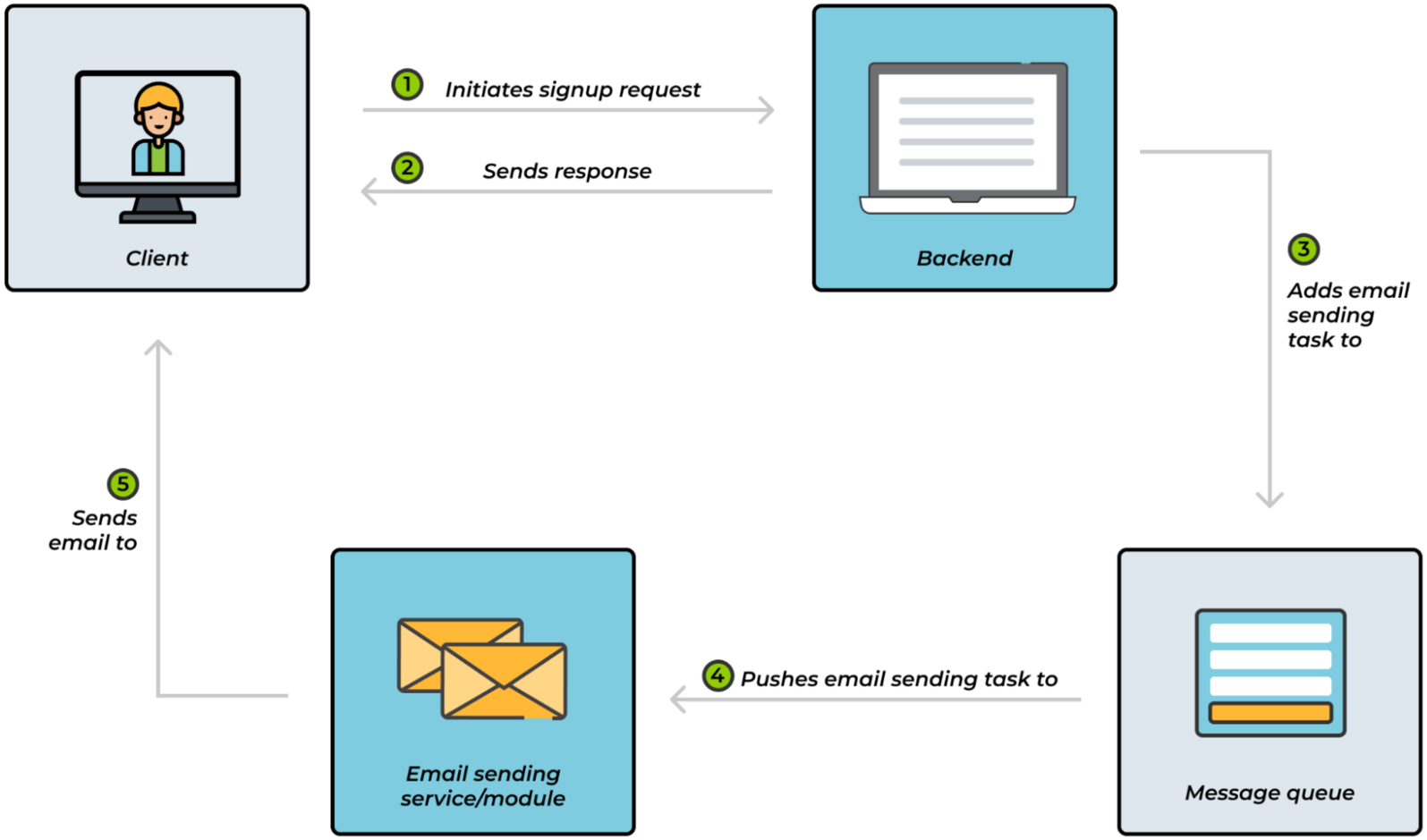


- اطلاعات session کاربر کجا باید ذخیره شود؟
- اگر اطلاعات درون سرور ذخیره شود، درخواست‌های بعدی کاربر باید به همان سرور ارسال شود (stateful).
- تمامی سرورها از یک پایگاه داده مشترک برای ذخیره اطلاعات session کاربر استفاده کنند (stateless).

02 Message Queue

Image Resizing

Email System

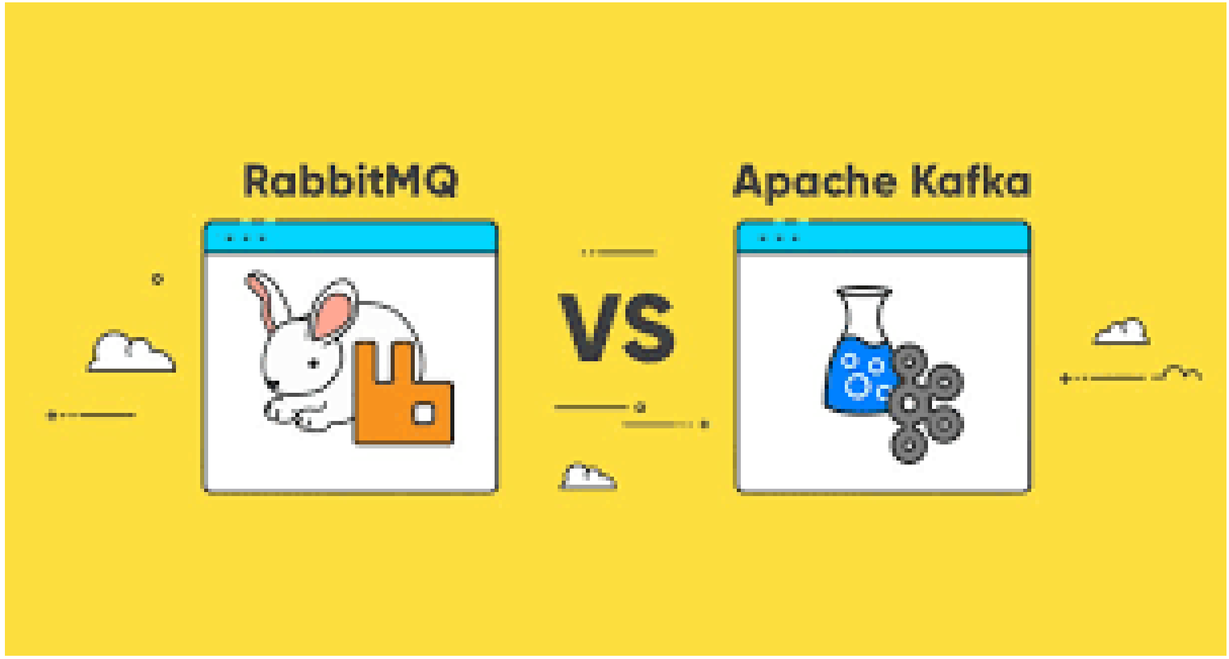


Message Queue

- صف پیغام (message queue) حداقل از سه عنصر تشکیل شده است:
  ۱. سرویسی که پیغام را تولید می‌کند (producer)
  ۲. یک صف که پیغام‌ها در آن قرار می‌گیرد (queue)
  ۳. سرویسی که پیغام را از صف می‌خواند (consumer)

- ارتباط بین دو سرویس ناهمگام (asynchronous) است.
- مقیاس‌پذیری با افزایش تعداد سرویس‌های consumer افزایش می‌یابد.
- اگر به هر دلیل سرویس consumer از کار بیافتد، کل سیستم همچنان کار می‌کند (fault tolerant).

Message Queue



03 Event Driven Architecture

Introduction to EDA

- معماری Event-drivenیا به اختصار EDAالگوی طراحی نرم‌افزار است که به سازمان‌ها اجازه می‌دهد «رویدادها» Event یا لحظات مهم در کسب‌وکارها (مثل تراکنش، بازدید سایت، ترک سبد خرید و غیره) را تشخیص داده و به آن‌ها در زمان واقعی یا نزدیک به زمان واقعی واکنش نشان دهند. این الگو جایگزین معماری سُنتی «درخواست/پاسخ» request/response می‌شود که در آن سرویس‌ها باید منتظر پاسخ می‌ماندند تا به کار بعدی بروند. جریان معماری Event-driven توسط eventها هدایت می‌شود و به گونه‌ای طراحی شده که به آن‌ها پاسخ داده یا عملی را در پاسخ به یک event انجام دهد.
- معماری Event-driven اغلب به‌عنوان ارتباط «asynchronous» یا «غیرهم‌زمان» شناخته می‌شود. یعنی فرستنده و گیرنده لازم نیست برای انجام کار بعدی منتظر یکدیگر بمانند و سیستم‌ها به یک پیام وابسته نیستند. مثلاً تماس تلفنی ارتباط «هم‌زمان» محسوب می‌شود، مثل معماری سنتی «درخواست/پاسخ»، که در آن درخواست‌دهنده منتظر پاسخ می‌ماند. یک مثال غیرهم‌زمان، پیامک است. شما پیامک می‌فرستید و ممکن است ندانید که گیرنده کیست یا آیا کسی پیام را می‌خواند، اما منتظر پاسخ نمی‌مانید.

Event Driven Architecture

- در چند سال گذشته، تمرکز از داده‌های ثابت ((Service Oriented Architectureبه سمت رویدادها ((Event-driven Architectureتغییر کرده است. به‌جای انباشتن داده‌ها و ایجاد Data Lake، اکنون به داده‌های در جریان و ردیابی آن‌ها در حرکت توجه می‌شود. سیستم‌های سنتی عمدتاً در یک مدل داده‌محور کار می‌کردند که داده‌ها منبع اصلی حقیقت بودند. تغییر به سمت معماری Event-driven به معنای حرکت از مدل داده‌محور به مدل رویدادمحور است.

- در مدل رویدادمحور، داده‌ها همچنان مهم هستند، اما eventها به مهم‌ترین بخش تبدیل می‌شوند. در مدل سرویس‌محور، اولویت اصلی حفظ داده‌ها بود، اما در معماری Event-driven، اولویت اصلی پاسخ به رویدادها به‌صورت همزمان است؛ زیرا ارزش رویدادها با گذشت زمان کاهش پیدا می‌کند. با این حال، امروزه معماری سرویس‌محور و معماری رویدادمحور اغلب به همراه هم استفاده می‌شوند.

What is Event?

- رویداد یا همان Eventبه تغییر وضعیت یک سیستم کلیدی کسب‌وکار گفته می‌شود. مثلاً، شخصی محصولی می‌خرد، فردی برای پرواز چک‌این می‌کند یا اتوبوسی دیر به مقصد می‌رسد. اگر دقت کنیم، Eventها در همه جا و در هر صنعتی رخ می‌دهند. هر چیزی که یک «پیام» ایجاد کند، از تولید و انتشار گرفته تا شناسایی و مصرف، به‌عنوان رویداد شناخته می‌شود. رویداد از پیام جداست، چون Event وقوع یک تغییر است و پیام، اعلان (نوتیفیکیشن) در حال حرکتِ این وقوع را انتقال می‌دهد. در معماری Event-driven، رویداد ممکن است یک یا چند عمل یا فرآیند را در پاسخ به وقوع خود تحریک ((Triggerکند. نمونه‌هایی از رویدادها شامل موارد زیر هستند:
  - درخواست برای بازنشانی رمز عبور
  - تحویل بسته به مقصد
  - به‌روزرسانی موجودی یک انبار مواد غذایی
  - رد تلاش برای دسترسی غیرمجاز
- هر یک از این رویدادها ممکن است یک یا چند عمل یا فرآیند را در پاسخ، Triggerکنند. یکی از پاسخ‌ها ممکن است به سادگیِ ثبت Eventبرای اهداف نظارتی باشد. پاسخ‌های دیگر می‌توانند شامل موارد زیر باشند:
  - ارسال ایمیل برای بازنشانی رمز عبور به مشتری
  - بسته شدن فروش بلیط
  - ثبت سفارش جدید برای کالاهای در حال اتمام مانند کاهو
  - قفل شدن حساب و اطلاع‌رسانی به پرسنل امنیتی
- در معماری Event-driven، زمانی که نوتیفیکیشن رویداد ارسال می‌شود، سیستم ثبت می‌کند که تغییری رخ داده است و منتظر می‌ماند تا پاسخ را به هر فردی که درخواست کرده، هر زمان که درخواست کند، ارسال کند. اپلیکیشنی که آن پیام را دریافت کرده می‌تواند بلافاصله پاسخ دهد یا تا زمانی که تغییر وضعیت مورد انتظار رخ دهد، منتظر بماند.
- اپلیکیشن‌های مبتنی بر معماری Event-drivenامکان ایجاد کسب‌وکارهای دیجیتال سریع‌تر، مقیاس‌پذیرتر، با زمینه بهتر و پاسخگوتر را فراهم می‌کنند.

Event Producer

- در معماری Event-driven، تولیدکننده‌های رویداد را داریم که نوتیفیکیشن‌های eventرا ایجاد و ارسال می‌کنند و ممکن است یک یا چند مصرف‌کننده eventهم وجود داشته باشد که در آنجا دریافت eventباعث اجرای منطق پردازش Processing Logic می‌شود.
- مثلاً فرض کنید نتفلیکس فیلم جدیدی را آپلود کرده است. چندین اپلیکیشن می‌توانند منتظر این نوتیفیکیشن باشند و سپس سیستم‌های داخلی خود را فعال کنند تا اطلاعات مربوط به آن eventرا به کاربرانشان بفرستند. این روش با پیام‌رسانی درخواست-پاسخ سنتی متفاوت است؛ در اینجا اپلیکیشن‌ها در حال اجرا هستند و حتی اگر منتظر این رویداد باشند، معطل نمی‌مانند و می‌توانند به محض دریافت پیام، پاسخ دهند. به این ترتیب، بسیاری از سرویس‌ها می‌توانند به‌صورت همزمان اجرا شوند.

Event Driven Architecture

- اجزای معماری رویدادمحور شامل ۳ بخش «تولیدکننده» ((Producer، »مصرف‌کننده» ((Consumerو «واسطه» ((Brokerاست. واسطه ممکن است اختیاری باشد، به‌خصوص زمانی که تولیدکننده و مصرف‌کننده مستقیماً با هم در ارتباط هستند و تولیدکننده eventها را به مصرف‌کننده می‌فرستد. به عنوان مثال، تولیدکننده می‌تواند رویدادها را فقط به یک پایگاه داده یا انبار داده ارسال کند تا رویدادها برای تحلیل جمع‌آوری و ذخیره شوند. معمولاً در شرکت‌ها، منابع متعددی وجود دارند که انواع مختلف رویدادها را ارسال می‌کنند و یک یا چند مصرف‌کننده به برخی یا همه آن رویدادها علاقه‌مند هستند.
- به یک مثال توجه کنیم. مثلاً اگر شما خرده‌فروش هستید، ممکن است تمامی خریدهایی را که در تمام فروشگاه‌های شما در سراسر جهان انجام می‌شوند، جمع‌آوری کنید. این رویدادها را که بر تقلب نظارت می‌کنند به معماری رویدادمحور خود می‌خورانید و آن‌ها را به پردازشگر کارت اعتباری ارسال می‌کنید یا هر اقدام مورد نیاز دیگر را انجام می‌دهید. برای تولیدکننده، شما انواع داده‌ها را از تجهیزات خود دریافت می‌کنید. این داده‌ها به شما اطلاعاتی مانند دما و فشار را می‌دهند تا بتوانید بسته به آنچه داده‌ها به شما می‌گویند، این رویدادها را به‌صورت لحظه‌ای نظارت کنید و اقداماتی مانند پیش‌بینی خرابی‌ها یا برنامه‌ریزی تعمیرات را انجام دهید.

04 Rate Limiter

Introduction

rate limiter چیست؟

- حداکثر ۳ بار تلاش ورود به سامانه در یک دقیقه گذشته
- حداکثر ارسال ۵۰ درخواست دوستی در روز
- حداکثر ایجاد ۲۰ پست در هر ساعت

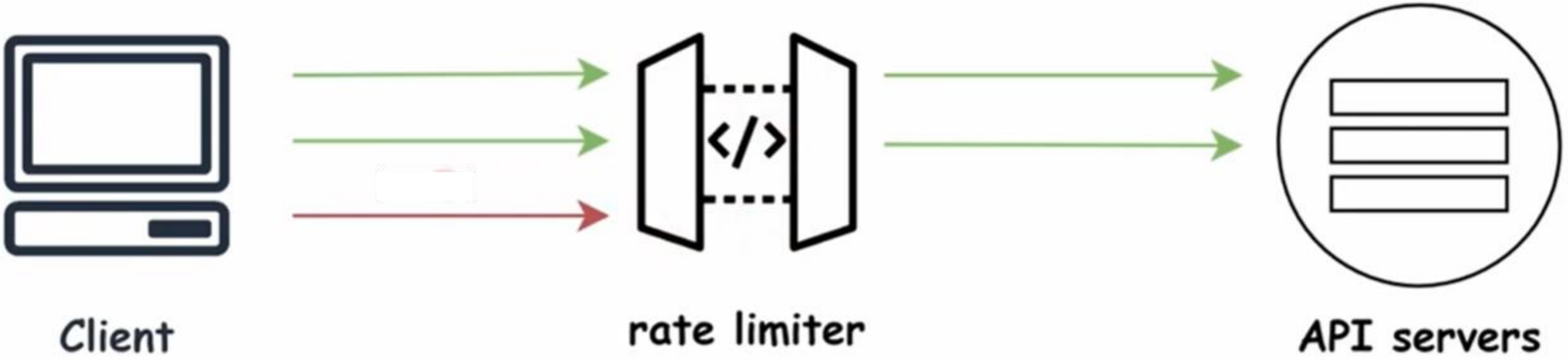
چرا به rate limiter احتیاج داریم؟

- مانع از حمله denial of service
- کاهش هزینه و بهره‌وری بالاتر

Rate Limiter Requirement

- محدودیت استفاده API ها به ازای شناسه کاربری یا IP
- سرعت بالا در اعمال محدودیت
- در صورت اعمال محدودیت پاسخی متناسب به کاربر داده شود

Rate Limiter



Response Sample

```
HTTP/1.1          429  Too Many Requests
Content-Type:     application/json
Retry-after:      53.29s
```

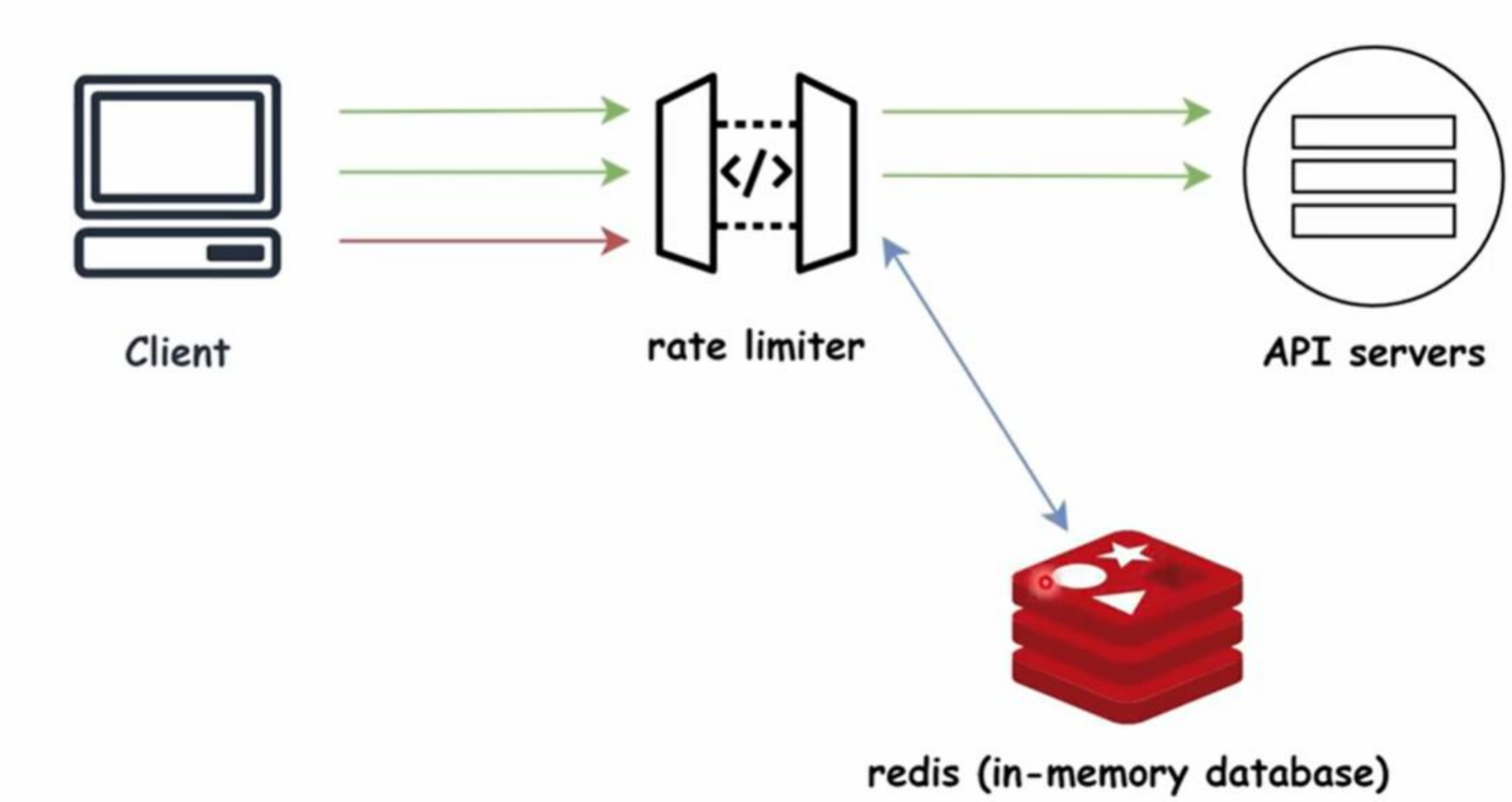
Requirement Data

- به آدرس IP یا شناسه کاربر
- تعداد درخواست‌های ارسالی کاربر به API در X دقیقه اخیر
- زمان آخرین درخواست (timestamp) به ازای هر کاربر

Token Bucket

- به ازای هر زوج مرتب کاربر و API یک سطل (bucket) در نظر می‌گیریم.
- هر سطل ظرفیت محدود از پیش تعیین شده دارد
- توکن‌ها با نرخ مشخص وارد سطل می‌شود
- اگر سطل پر شود دیگر توکنی وارد آن نمی‌شود
- هر درخواست یک توکن از سطل بر می‌دارد
- اگر به اندازه کافی توکن در سطل وجود داشته باشد، درخواست پذیرفته می‌شود. در غیر این صورت درخواست رد می‌شود.

Rate Limiter Design



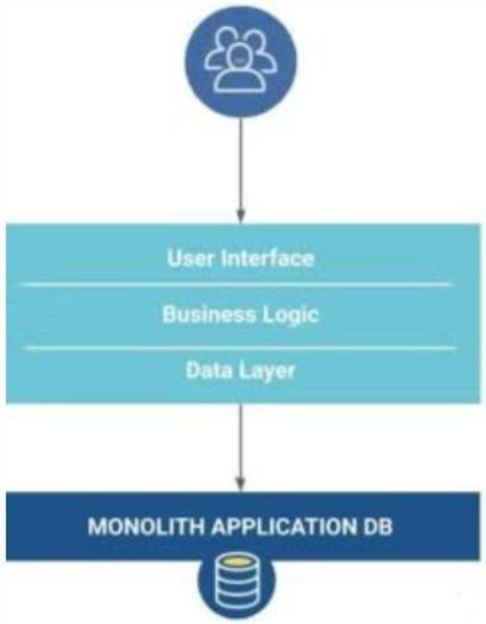
Questions

- چگونه برای سیستم توزیع شده آن را توسعه دهیم؟
- چگونه الگوریتم را قابل پیکربندی کنیم؟
- چگونه قوانین خاص را برای کاربر مورد نظر اعمال کنیم؟

05 Monolithic Architecture

Monolithic Architecture

- Monolithic architecture refers to a single unified codebase where all components of an application are interconnected and interdependent. In this model, the entire application is built as one unit, making it easier to develop and deploy initially.
- Characteristics of Monolith Architecture:
  - Single Codebase: All application components reside within a single codebase, simplifying development and deployment.
  - Tightly Coupled: Components are highly interconnected, making changes to one part of the application potentially affect others.
  - Easier to Test: Integrated testing can be straightforward, as everything is in one place.
  - Simpler Deployment: A single deployment process is required, reducing complexity.



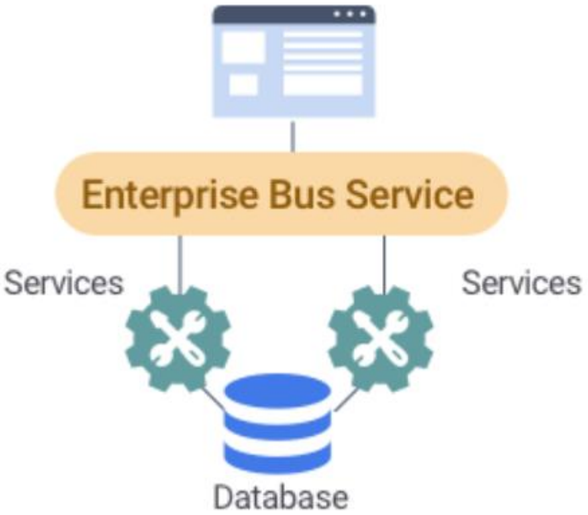
Use Cases of Monolith Architecture

- Small Applications: Ideal for small-scale applications where complexity is low and a quick development cycle is needed.
- Startups: Many startups begin with a monolithic approach to quickly build and validate their product before scaling.
- Legacy Systems: Existing applications that have not transitioned to newer architectures often remain monolithic.

06 Service-Oriented Architecture (SOA)

SOA

- Service-Oriented Architecture is an architectural pattern that structures an application as a collection of loosely coupled services. Each service represents a specific business function and can be independently developed, deployed, and scaled.



- Characteristics of SOA:
  - Loose Coupling: Services interact through well-defined interfaces, minimizing dependencies between them.
  - Interoperability: SOA enables different services to communicate, regardless of the technology stack used.
  - Reusability: Services can be reused across different applications, improving development efficiency.
  - Scalability: Individual services can be scaled independently based on demand.
  - Storage: SOA architecture typically includes a single data storage layer that is shared by all services within a given application

SOA Limitations

- The enterprise service bus (ESB) connects multiple services together, which makes it a single point of failure.
- SOA architecture typically includes a single data storage layer that is shared by all services. So, data consistency and integrity are critical challenges in SOA.
- All services share a common data repository. This makes the services difficult to manage individually.
- Every service has a broad scope. So, if one of the services fails, the entire business workflow will be affected.
- Consistent data governance across all services.
- Slows down as more services are added on.
- Managing service versioning and compatibility

Use Cases of Service-Oriented Architecture (SOA)

- Enterprise Applications: Large organizations often adopt SOA to integrate various systems and improve interoperability between departments.
- Integration Scenarios: SOA is effective for integrating legacy systems with new applications, allowing for gradual modernization.
- Business Process Management: Companies that require dynamic and adaptable business processes can benefit from SOA's flexibility.

07 Microservices Architecture

Microservice

- Microservices architecture is a more granular approach to SOA, where an application is broken down into smaller, independently deployable services. Each microservice focuses on a specific business capability and communicates with others via APIs.
- Characteristics of Microservices Architecture:
  - Highly Decoupled: Microservices are designed to function independently, allowing for isolated development and deployment.
  - Focused Functionality: Each **microservice serves a specific business capability**, enabling teams to work on different features simultaneously.
  - Polyglot Persistence: Different microservices can use **different databases and technologies**, allowing for flexibility in design.
  - Continuous Delivery: Microservices can be deployed independently, enabling faster updates and continuous integration.

Use Cases of Microservices Architecture

- E-commerce Platforms: Companies like Amazon utilize microservices to handle various functions (e.g., inventory, payment) independently, allowing for rapid updates and scalability.
- Streaming Services: Platforms like Netflix leverage microservices to provide scalable and resilient services that can handle millions of users simultaneously.
- Continuous Deployment: Organizations that need to frequently update and deploy features can use microservices to enhance agility and reduce downtime.

Choosing the right architecture depends on project requirements, team capabilities, and future scalability.

Comparison

Property	SOA	Microservices
Dependency	Business units are dependent	Each microservice is independent, and they don't usually share a data layer
Design	Aims to maximize application service reusability	Focused on decoupling
Connection	Basic form of communication is by Enterprise Event Bus	Services communicate via message protocols and sometimes by REST
Software size	Usually larger than in any other architecture	Consist of small independent service
Architecture	Fine-grained or centralized	Coarse-grained or decentralized
Services	Business, enterprise or infrastructure	Functional and infrastructure

Conclusion

- Q1: What are the main advantages of monolithic architecture?

  Simple deployment, easier testing, and faster initial development.
- Q2: How does SOA differ from microservices architecture?

  SOA consists of larger, loosely coupled services, while microservices are smaller and designed to operate independently.
- Q3: What challenges are associated with microservices?

  Increased complexity in service management, communication, and potential for distributed system issues.
- Q4: Is it possible to convert a monolith to microservices?

  Yes, this process is often called "decomposition," but it can be complex and requires careful planning.
- Q5: When should I choose SOA over microservices?

  SOA is ideal for organizations needing integration across various systems without the complexity of microservices.

References

- https://hamravesh.com/blog/what-is-event-driven-architecture/