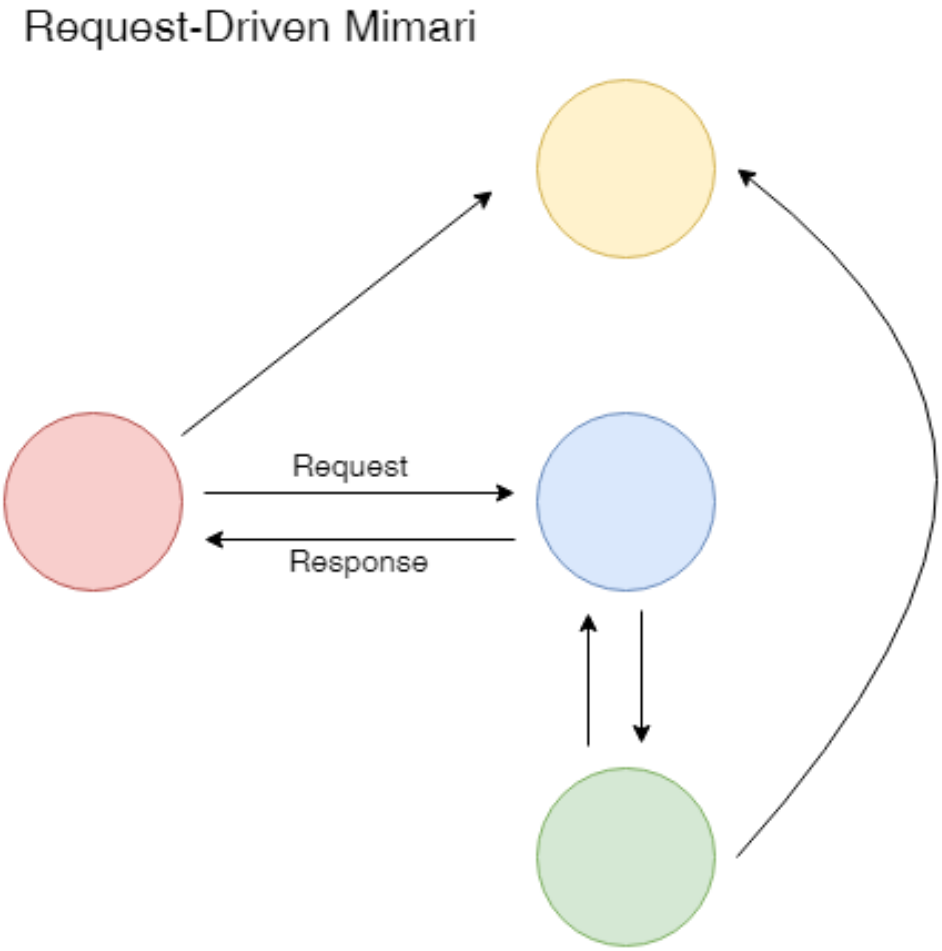
Senkron - Asenkron İletişim

Request-Driven Mimari
Servis sayımız arttıkça mimarimizin karmaşıklığı da, http trafiğimiz de doğru orantılı olarak artacaktır. Bir microservice, datasına ihtiyaç duyduğu başka bir servise, bir http client üzerinden istekte bulunur ve bu işlem bir IO (Thread Blocking IO) işlemi olduğu için aslında maliyetli de bir işlemdir. Servisler (sunucular) client'larıyla olan bu iletişimi socket'ler üzerinden yaparlar ve bu socket'ler sonsuz sayıda değildir, bu yüzden socket kullanımı iyi yönetilmesi gereken bir konudur.



Bu mimari de adından da anlaşılacağı üzere bir service, verisine ihtiyaç duyduğu bir diğer servise doğrudan istekte bulunur. Servislerimizin modern Rest servisleri olduğunu kabul edersek yapılan istekler, GET, POST, DELETE ve UPDATE isteklerinden ibaret olacaktır.

Senkron (Synchronous) İletişim
Http protokolü senkron çalışan bir protokoldür. Client bir istek yapar ve sunucudan yanıt dönmesini bekler. Client tarafında servis çağrısını yapan kodun senkron (thread'in

blocklanması durumu) veya asenkron (thread'in bloklanmaması ve yanıtın call back ile gelmesi) yazılması Http'nin senkron bir protokol olduğu gerçeğini değiştirmez.

Asenkron (Asynchronous) İletişim
Http'nin senkron çalıştığından ve call back mekanizmalarıyla bir şekilde client veya sunucu tarafında bir asenkronizasyon şeklinde çalışır.

Spring @Async Kullanımı:
https://spring.io/guides/gs/async-method/

```
  @Async
  public CompletableFuture<User> findUser(String user) throws
InterruptedException {
    logger.info("Looking up " + user);
    String url = String.format("https://api.github.com/users/%s",
user);
    User results = restTemplate.getForObject(url, User.class);
    // Artificial delay of 1s for demonstration purposes
    Thread.sleep(1000L);
    return CompletableFuture.completedFuture(results);
  }
```

```
@SpringBootApplication
@EnableAsync
public class AsyncMethodApplication {

  public static void main(String[] args) {
    // close the application context to shut down the custom
ExecutorService
    SpringApplication.run(AsyncMethodApplication.class,
args).close();
  }

  @Bean
  public Executor taskExecutor() {
    ThreadPoolTaskExecutor executor = new
ThreadPoolTaskExecutor();
    executor.setCorePoolSize(2);
    executor.setMaxPoolSize(2);
    executor.setQueueCapacity(500);
    executor.setThreadNamePrefix("GithubLookup-");
```

```
    executor.initialize();
    return executor;
  }


}
```

```
@Component
public class AppRunner implements CommandLineRunner {

  private static final Logger logger =
LoggerFactory.getLogger(AppRunner.class);

  private final GitHubLookupService gitHubLookupService;

  public AppRunner(GitHubLookupService gitHubLookupService)
{
    this.gitHubLookupService = gitHubLookupService;
  }

  @Override
  public void run(String... args) throws Exception {
    // Start the clock
    long start = System.currentTimeMillis();

    // Kick of multiple, asynchronous lookups
    CompletableFuture<User> page1 =
gitHubLookupService.findUser("PivotalSoftware");
    CompletableFuture<User> page2 =
gitHubLookupService.findUser("CloudFoundry");
    CompletableFuture<User> page3 =
gitHubLookupService.findUser("Spring-Projects");

    // Wait until they are all done
    CompletableFuture.allOf(page1,page2,page3).join();

    // Print results, including elapsed time
    logger.info("Elapsed time: " +
(System.currentTimeMillis() - start));
    logger.info("--> " + page1.get());
    logger.info("--> " + page2.get());
    logger.info("--> " + page3.get());

  }
```