

JAVA BACKEND DEVELOPMENT PROGRAM

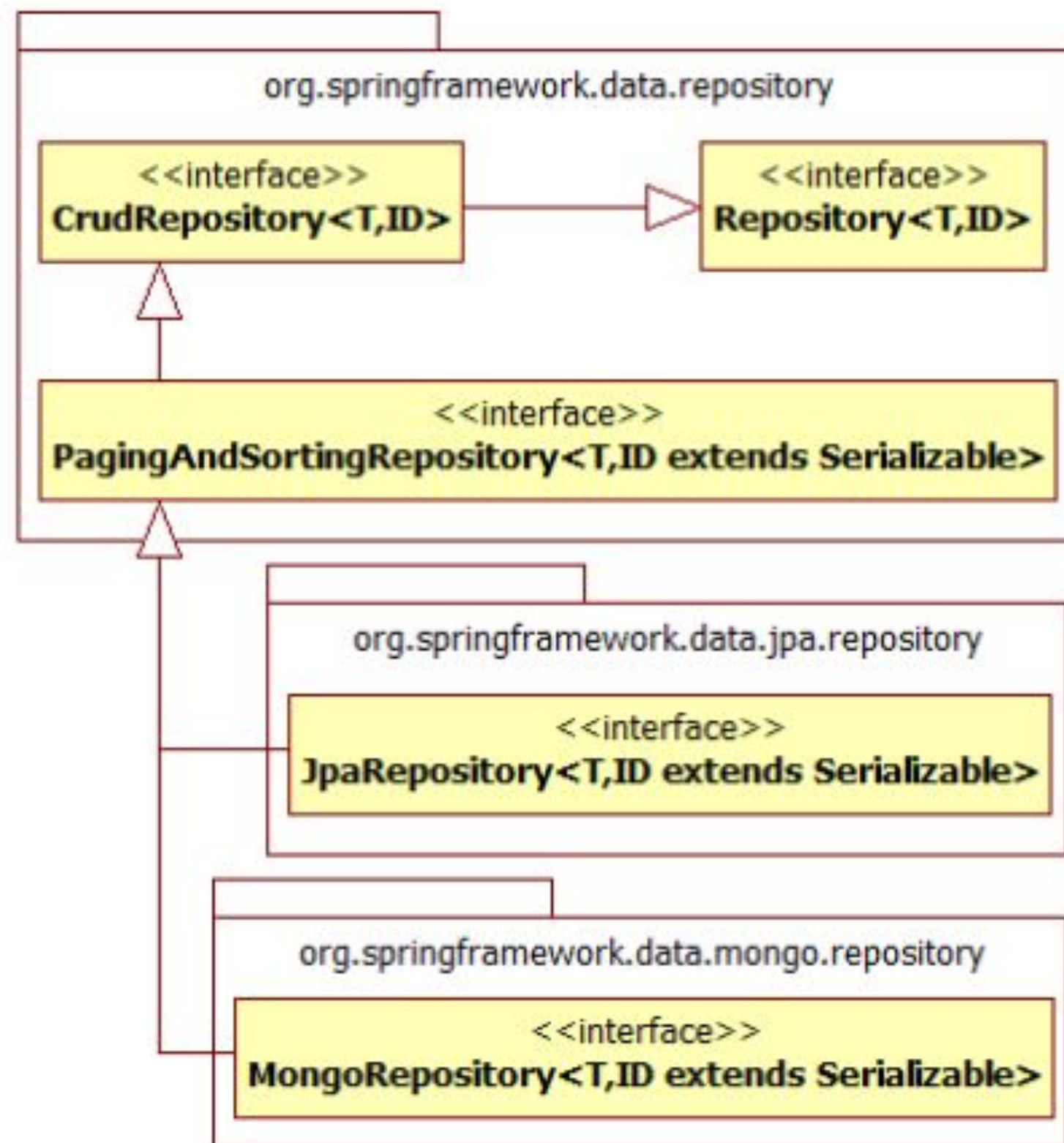
Spring Data Repository, Caching, Swagger

OUTLINE

- Spring Data Repository
 - Core Concept
 - Query
 - MongoDB Integration
- Caching
 - Introduction
 - Spring Cache
 - Redis
- Swagger

SPRING DATA REPOSITORY

- The central interface in the Spring Data repository abstraction is *Repository*.
- It takes the domain class to manage as well as the ID type of the domain class as type arguments.
- This interface acts primarily as a *marker interface* to capture the types to work with and to help you to discover interfaces that extend this one.



MARKER INTERFACE

- A marker interface is an interface that has no methods or constants inside it.
- It provides run-time type information about objects, so the compiler and JVM have additional information about the object
 - *Serializable* and *Cloneable* are used to indicate something to compiler or JVM.
 - So if JVM sees a Class is *Serializable* it does some special operation on it, similar way if JVM sees one Class is implement *Cloneable* it performs some operation to support cloning

CRUDREPOSITORY

- The *CrudRepository* provides sophisticated CRUD functionality for the entity class that is being managed

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {  
    <S extends T> S save(S entity);           1  
    Optional<T> findById(ID primaryKey);      2  
    Iterable<T> findAll();                     3  
    long count();                             4  
    void delete(T entity);                     5  
    boolean existsById(ID primaryKey);         6  
    // ... more functionality omitted.  
}
```

- Note that the ID here is a serializable which represent the id of the Entity

PAGINGANDSORTINGREPOSITORY

- The *PagingAndSortingRepository* abstraction that adds additional methods to ease paginated access to entities

```
PagingAndSortingRepository<User, Long> repository = // ... get access to a bean  
Page<User> users = repository.findAll(PageRequest.of(1, 20));
```

- To access the second page of User by a page size of 20

```
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {  
    Iterable<T> findAll(Sort sort);  
    Page<T> findAll(Pageable pageable);  
}
```

- A *Page<T>* instance, in addition to having the list of *Products*, also knows about the total number of available pages. It triggers an additional count query to achieve it.
- To avoid such an overhead cost, we can instead return a *Slice<T>* or a *List<T>*.
- A *Slice* only knows about whether the next slice is available or not.

QUERY

- The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository.
- The mechanism strips the prefixes from the method and starts parsing the rest of it:
 - find...By (May return empty result)
 - read...By
 - query...By
 - count...By
 - get...By (If not existed, then throw an exception)
- The introducing clause can contain further expressions, such as a *Distinct* to set a distinct flag on the query to be created
- However, the first *By* acts as delimiter to indicate the start of the actual criteria

QUERY

- We also get support for operators such as *Between*, *LessThan*, *GreaterThan*, and *Like* for the property expressions. The supported operators can vary by datastore
- The method parser supports setting an *IgnoreCase* flag for individual properties (for example, *findByLastnameIgnoreCase(...)*) or for all properties of a type that supports ignoring cases
- We can apply static ordering by appending an *OrderBy* clause to the query method that references a property and by providing a sorting direction (*Asc* or *Desc*)

QUERY

```
interface PersonRepository extends Repository<Person, Long> {  
  
    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);  
  
    // Enables the distinct flag for the query  
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);  
  
    // Enabling ignoring case for an individual property  
    List<Person> findByLastnameIgnoreCase(String lastname);  
    // Enabling ignoring case for all suitable properties  
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);  
  
    // Enabling static ORDER BY for a query  
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);  
}
```


QUERY

- Property Expressions
 - Assume a Person has an Address with a ZipCode.
 - `List<Person> findByAddressZipCode(ZipCode zip)`
 - In that case, the method creates the property traversal `x.address.zipCode`.
 - Although this should work for most cases, it is possible for the algorithm to select the wrong property. (Suppose the Person class has an `addressZip` property as well)
 - `List<Person> findByAddress_ZipCode(ZipCode zip)`
 - To resolve this ambiguity we can use the **underscore symbol** `_` inside the method name to manually define traversal points

QUERY

- Limiting Query Results
 - The results of query methods can be limited by using the first or top keywords, which can be used interchangeably.
 - An optional numeric value can be appended to top or first to specify the maximum result size to be returned
 - If the number is left out, a result size of 1 is assumed

```
User findFirstByOrderByLastnameAsc();  
User findTopByOrderByAgeDesc();  
Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);  
Slice<User> findTop3ByLastname(String lastname, Pageable pageable);  
List<User> findFirst10ByLastname(String lastname, Sort sort);  
List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

- Note: Limiting the results in combination with dynamic sorting by using a Sort parameter lets you express query methods for the 'K' smallest as well as for the 'K' biggest elements.

QUERY

- `@Query`
- Queries annotated to the query method take precedence over queries defined using `@NamedQuery` or named queries declared in `orm.xml`

```
public interface UserRepository extends JpaRepository<User, Long> {  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmailAddress(String emailAddress);  
}
```

- The `@Query` annotation allows for running native queries by setting the `nativeQuery` flag to true

```
public interface UserRepository extends JpaRepository<User, Long> {  
    @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery = true)  
    User findByEmailAddress(String emailAddress);  
}
```

QUERY

- Customizing Repository
- To enrich a repository with custom functionality, you must first define a fragment interface and an implementation for custom functionality

```
interface CustomizedUserRepository {  
    void someCustomMethod(User user);  
}
```

```
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {  
  
    public void someCustomMethod(User user) {  
        // Your custom implementation  
    }  
}
```

```
interface UserRepository extends CrudRepository<User, Long>, CustomizedUserRepository {  
  
    // Declare query methods here  
}
```


QUERY

- To use `NamedQuery` or `NativeQuery` is not type-safe, as we will only know if the query is wrong at runtime.
- *Querydsl* is a framework that enables the construction of statically typed SQL-like queries through its fluent API
- Several Spring Data modules offer integration with *Querydsl* through *QuerydslPredicateExecutor*

```
public interface QuerydslPredicateExecutor<T> {  
    Optional<T> findById(Predicate predicate); 1  
    Iterable<T> findAll(Predicate predicate); 2  
    long count(Predicate predicate); 3  
    boolean exists(Predicate predicate); 4  
    // ... more functionality omitted.  
}
```

QUERY

- To make use of Querydsl support, extend QuerydslPredicateExecutor on your repository interface

```
interface UserRepository extends CrudRepository<User, Long>, QuerydslPredicateExecutor<User> {  
}
```

- So type-safe queries can be used

```
Predicate predicate = user.firstname.equalsIgnoreCase("dave")  
    .and(user.lastname.startsWithIgnoreCase("mathews"));  
  
userRepository.findAll(predicate);
```


SQL VS NOSQL

- Schema
- Scalability
- Query
- Transaction

SQL VS NOSQL

- **Schema:** SQL databases have a rigid schema whereas NoSQL databases don't.
- **Scalability:** NoSQL databases are generally more horizontally scalable compared to SQL databases due to storing data as self-contained documents.
- **Query:** SQL syntax vs NoSQL database-dependent querying methods.
- **Transaction:** NoSQL databases generally do not support transaction across multiple documents whereas SQL databases do.

PARTITIONING

- Vertical Partitioning
 - id | name | age | hobbies
 - id is the partition key
 - id | name -> server 1
 - id | age | hobbies -> server 2
- Horizontal Partitioning
 - Let's say we have one billion rows
 - id is the partition key
 - 500 million rows -> server 1
 - 500 million rows -> server 2

NOSQL DATABASES

- MongoDB: General
- Redis: caching (key-value)
- GraphQL: REST APIs
- Cassandra: write fast (Wide Column)
- Elastic Search: indexing (Searching)
- <https://db-engines.com/en/ranking>

MONGODB

- MongoDB is a document database with the scalability and flexibility that you want with the querying and indexing that you need
- Three main components in MongoDB
 - Database — In MongoDB, databases hold collections of documents.
 - Collections — MongoDB stores documents in collections. Collections are analogous to tables in relational databases
 - Document — MongoDB stores data records as BSON documents. BSON is a binary representation of JSON documents

SQL

DB

Person Table		Column
Id	Name	Address
1 PK	Mueller	1 Row
2	<null>	2 FK

Address Relation

Id	City	Street
1	Leipzig	Burgstr. 1
2	Dresden	<null>

MongoDB

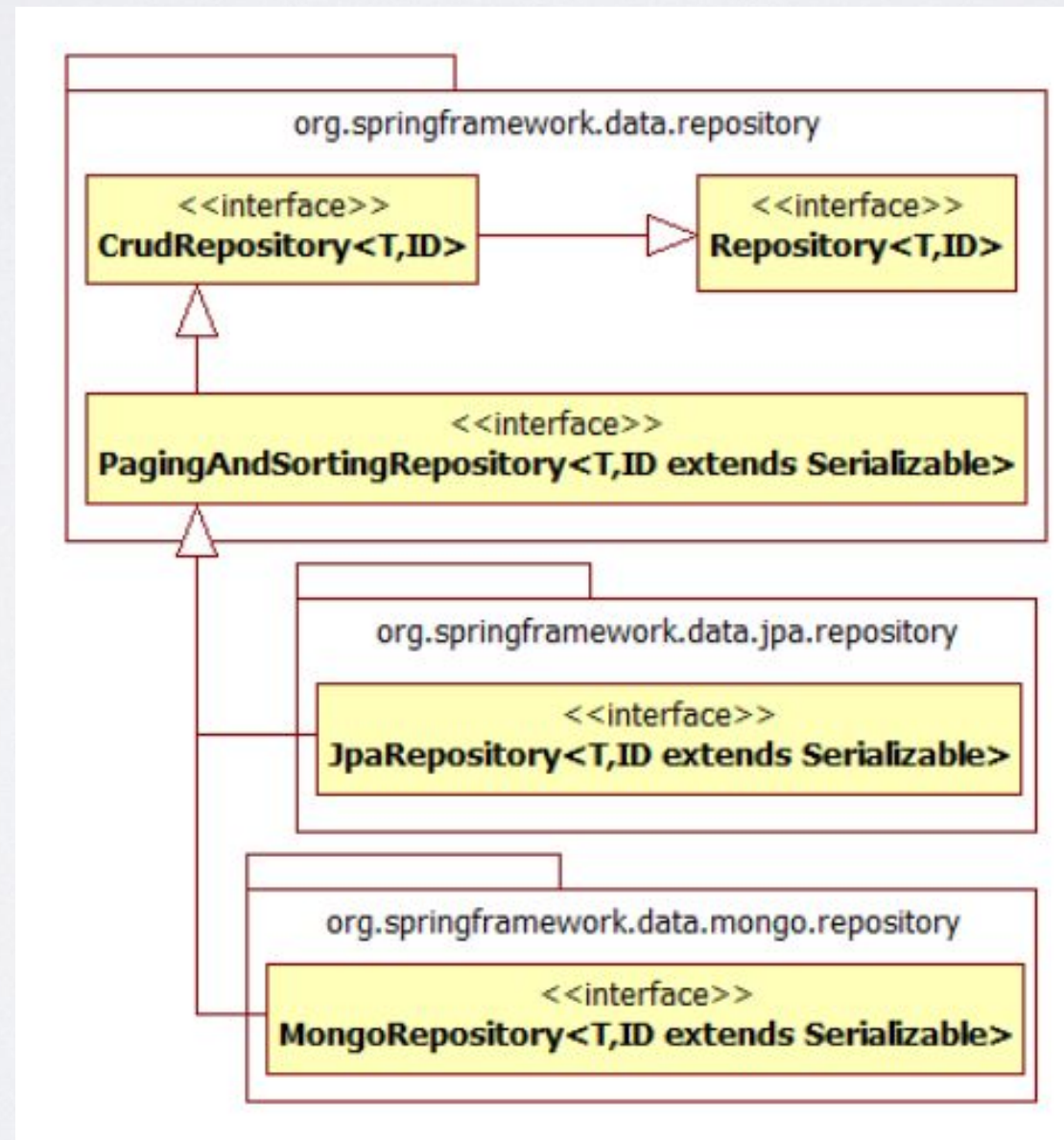
DB

Person Collection

```
{ Document
  _id: ObjectId("..."), Field
  Name: "Mueller", Key: Value
  Address: {
    City: "Leipzig",
    Street: "Burgstr. 1",
  }, Embedded document
}, {
  _id: ObjectId("..."), PK
  Address: {
    City: "Leipzig",
  },
}
```

MONGODB INTEGRATION

- MongoRepository is another extension of Spring Data Repository we explained before, which means all the functions can be applied to it.



MONGODB REPOSITORY

- Unlike our NamedQuery or NativeQuery which are supported by JpaRepository, MongoRepository use JSON-based query.

```
public interface PersonRepository extends MongoRepository<Person, String>

    @Query("{ 'firstname' : ?0 }")
    List<Person> findByThePersonsFirstname(String firstname);

}
```

- If we just want specific fields to be selected, we can use fields properties in *@Query* annotation

```
public interface PersonRepository extends MongoRepository<Person, String>

    @Query(value="{ 'firstname' : ?0 }", fields="{ 'firstname' : 1, 'lastname' : 1}")
    List<Person> findByThePersonsFirstname(String firstname);

}
```

SPRING CACHE

- To get started, we need following dependency

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-cache</artifactId>  
</dependency>
```

- Then we have to enable caching by **@EnableCaching** annotation
 - The **@EnableCaching** annotation triggers a post-processor that inspects every Spring bean for the presence of caching annotations on public methods
 - If such an annotation is found, a proxy is automatically created to intercept the method call and handle the caching behavior accordingly
- After we enable caching – for the minimal setup – we must register a **cacheManager**
 - Spring Boot automatically configures a suitable **CacheManager** to serve as a provider for the relevant cache (By default, it uses a **ConCurrentHashMap**)

SPRING CACHE

- `@Cacheable` — Defines methods that are cacheable so that on subsequent invocations (with the same arguments), the value in the cache is returned without having to actually invoke the method
 - The `findBook` method is associated with the cache named `books`

```
@Cacheable("books")  
public Book findBook(ISBN isbn) {...}
```

- `@CachePut` — When the cache needs to be updated without interfering with the method execution, you can use the `@CachePut` annotation

```
@CachePut(cacheNames="book", key="#isbn")  
public Book updateBook(ISBN isbn, BookDescriptor descriptor)
```

- It supports the same options as `@Cacheable`
- Using `@CachePut` and `@Cacheable` annotations on the same method is generally strongly discouraged because they have different behaviors
 - `@Cacheable` causes the method execution to be skipped by using the cache, but `@CachePut` forces the execution in order to execute a cache update, resulting in conflicting behaviours

SPRING CACHE

- `@CacheEvict` — This process is useful for removing stale or unused data from the cache
 - Using the `allEntries` attribute to evict all entries from the cache.

```
@CacheEvict(cacheNames="books", allEntries=true)
public void loadBooks(InputStream batch)
```

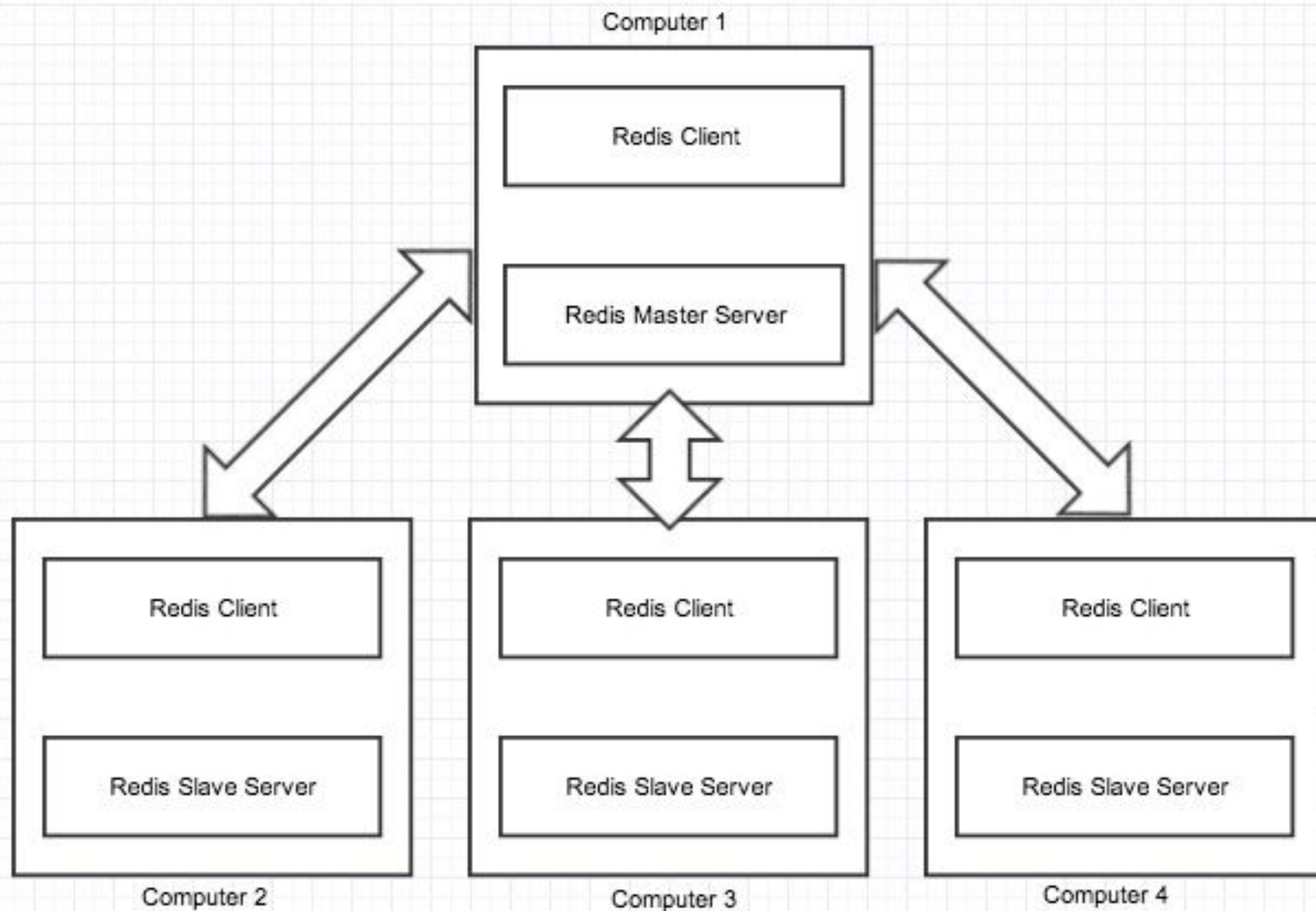
- `@Caching` — Sometimes, multiple annotations of the same type (such as `@CacheEvict` or `@CachePut`) need to be specified

```
@Caching(evict = { @CacheEvict("primary"), @CacheEvict(cacheNames="secondary", key="#p0") })
public Book importBooks(String deposit, Date date)
```


REDIS

- Redis is an open source, in-memory data structure store used as a database, cache and message broker
- In order to achieve its outstanding performance, Redis works with an in-memory dataset.
 - Depending on our use case, we can persist it either by dumping the dataset to disk every once in a while, or by appending each command to a log
- Redis also supports trivial-to-setup master-slave asynchronous replication, with very fast non-blocking first synchronization, auto-reconnection with partial resynchronization on net split
 - Replication is a technique involving many computers to enable fault-tolerance and data accessibility.
 - In a replication environment many computers share the same data with each other so that even if few computers go down, all the data will be available

REDIS



This image shows a basic Redis replication

REDIS

- All the slaves contain exactly same data as master. There can be as many as slaves per master server. When a new slave is inserted to the environment, the master automatically syncs all data to the slave
- If a slave fails, then also the environment continues working. when the slave again starts working, the master sends updated data to the slave.
- If there is a crash in master server and it loses all data then you should convert a slave to master instead of bringing a new computer as a master.

Docker

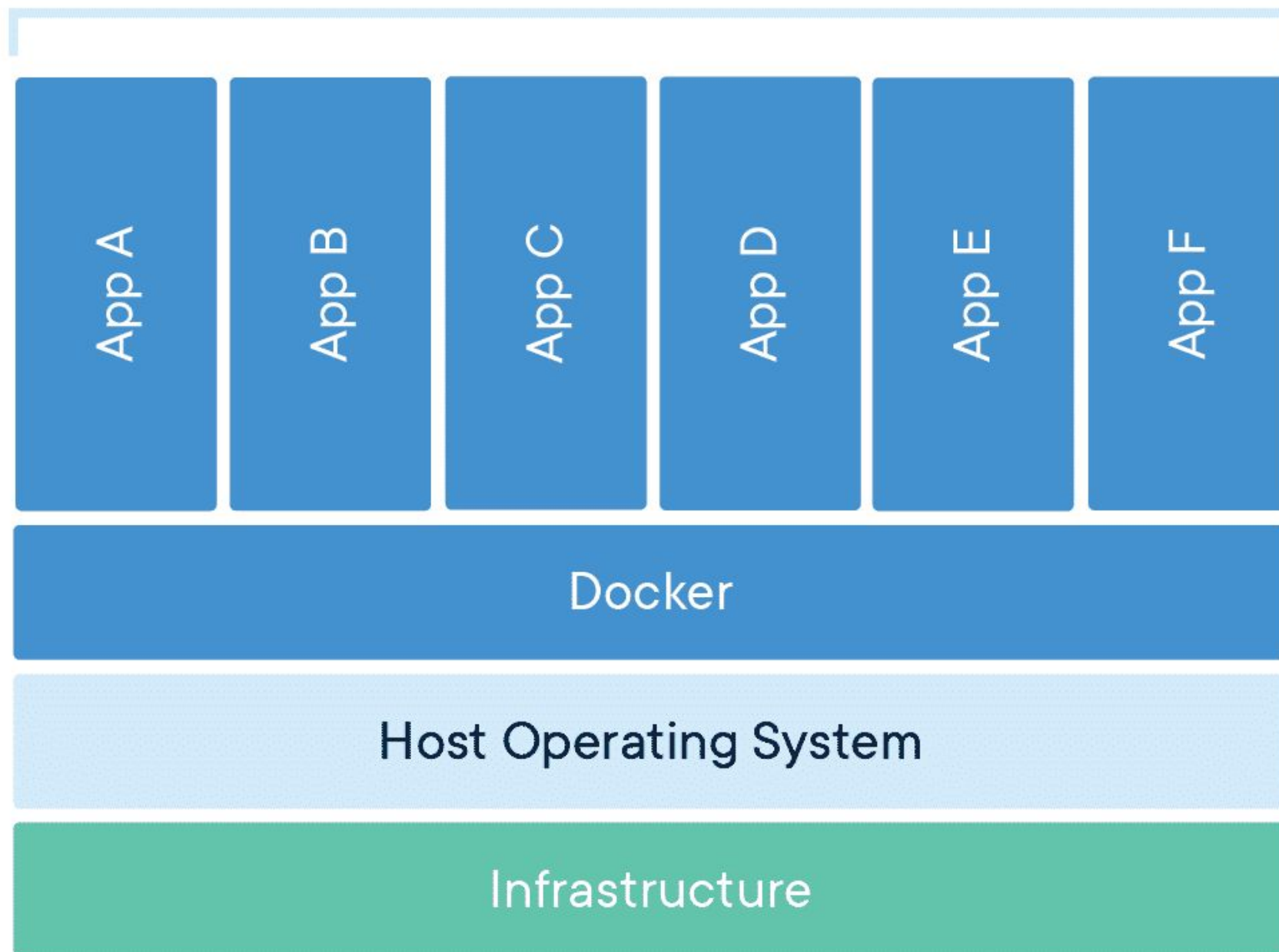
Docker is a standalone software that can be installed on any computer to run containerized applications. Containerization is an approach of running applications on an OS such that the application is isolated from the rest of the system.

Docker allows users to publish docker images and consume those published by others in repositories like Docker Hub.

- image: A static template - a set of bytes (class)
- container: Running version of your image, a image can have several containers (object)

Docker


Containerized Applications



SWAGGER

- Nowadays, front-end and back-end components often separate a web application.
- Usually, we expose APIs as a back-end component for the front-end component or third-party app integrations.
- In such a scenario, it is essential to have proper specifications for the back-end APIs.
- At the same time, the API documentation should be informative, readable, and easy to follow.
- Moreover, reference documentation should simultaneously describe every change in the API.
- Accomplishing this manually is a tedious exercise, so automation of the process was inevitable

SWAGGER

 **Swagger.**
Powered by SMARTBEAR

/api-docs

Explore

OpenAPI definition

v0 OAS3

/api-docs

Servers

http://localhost:8080 - Generated server url

default

⌵

GET

/api/book/{id}

PUT

/api/book/{id}

DELETE

/api/book/{id}

PATCH

/api/book/{id}

GET

/api/book/

POST

/api/book/

QUESTIONS?