Web Frameworks: Spring

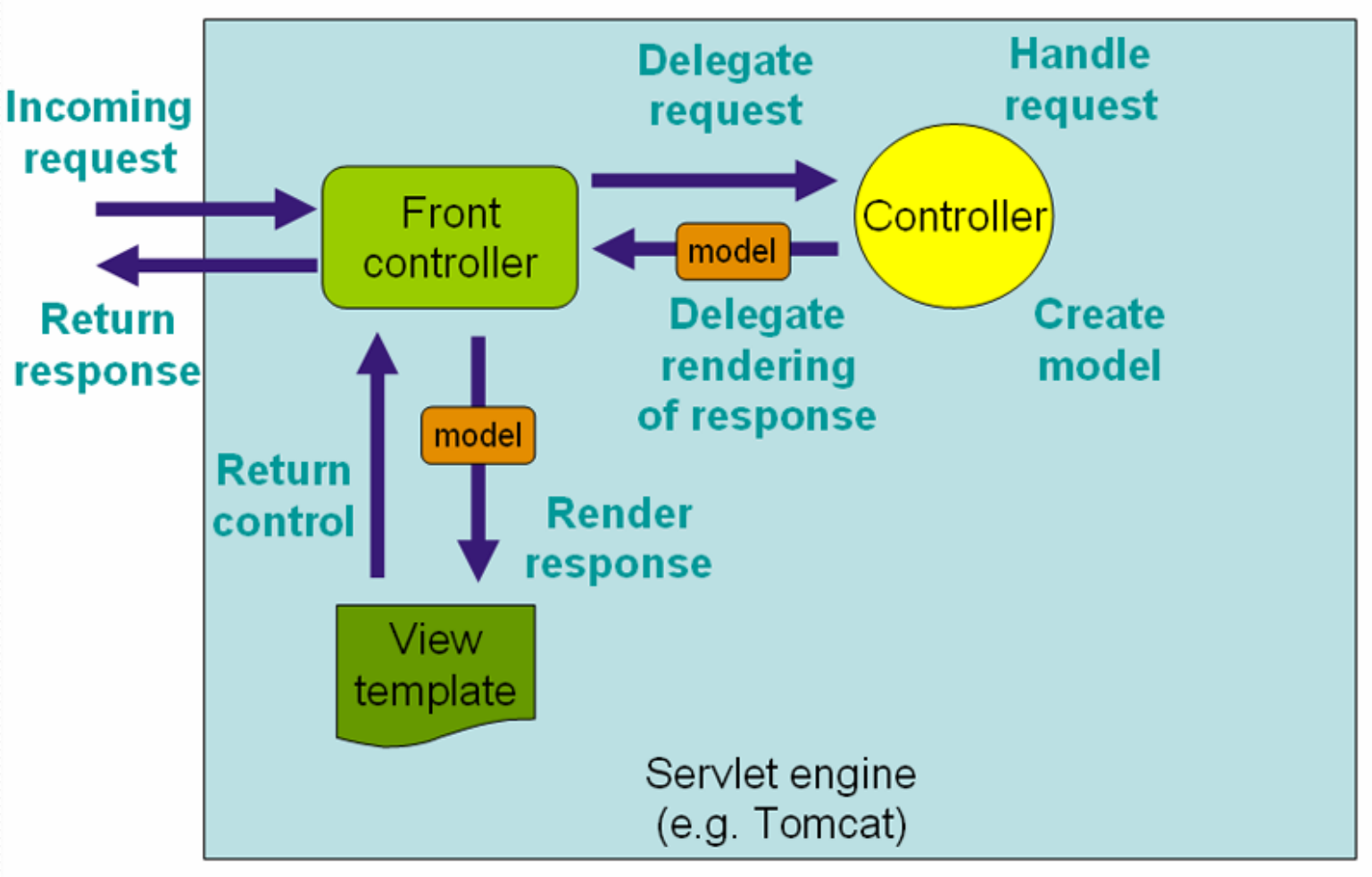
An Introduction

ApplicationContext

- If Dependency Injection is the core concept of Spring, then the ApplicationContext is its core object.
- The interface `org.springframework.context.ApplicationContext` represents the Spring IoC container
- It extends the `BeanFactory` interface, in addition to extending other interfaces to provide additional functionality in a more *application framework-oriented style*
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata
- Several implementations of the `ApplicationContext` interface are supplied out-of-the-box with Spring
  - such as ContextLoader that automatically instantiates an `ApplicationContext` as part of the normal startup process

https://docs.spring.io/spring-framework/docs/3.0.0.M4/spring-framework-reference/html/ch15s02.html

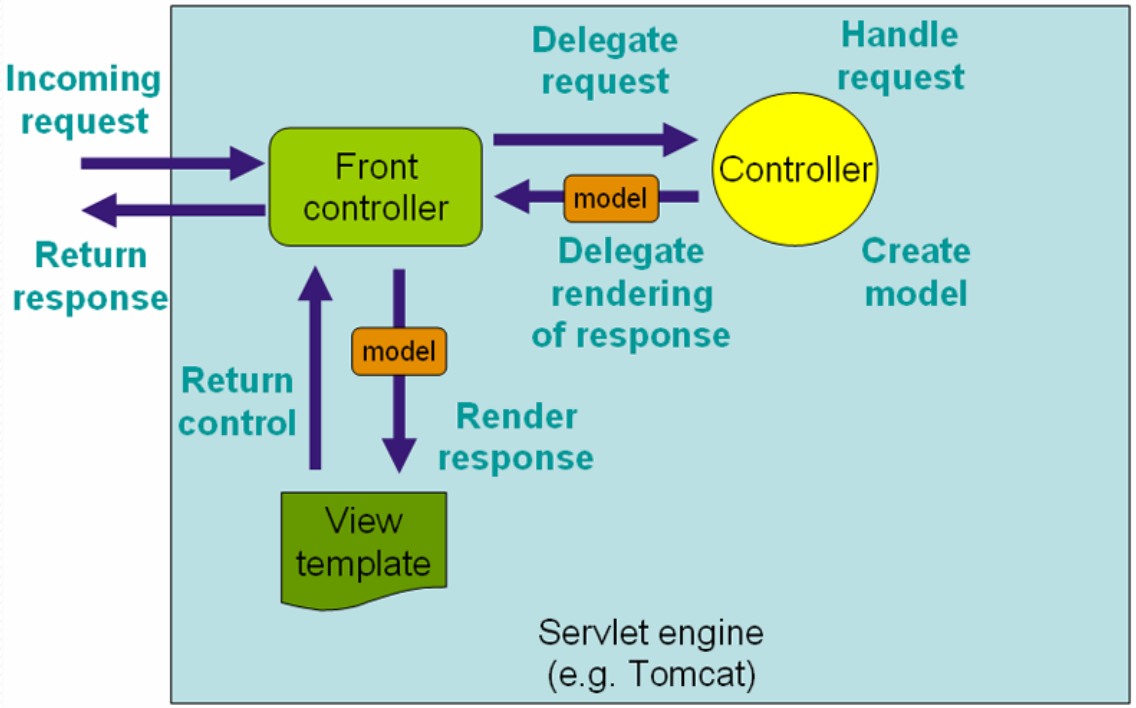
MVC Workflow



DispatcherServlet

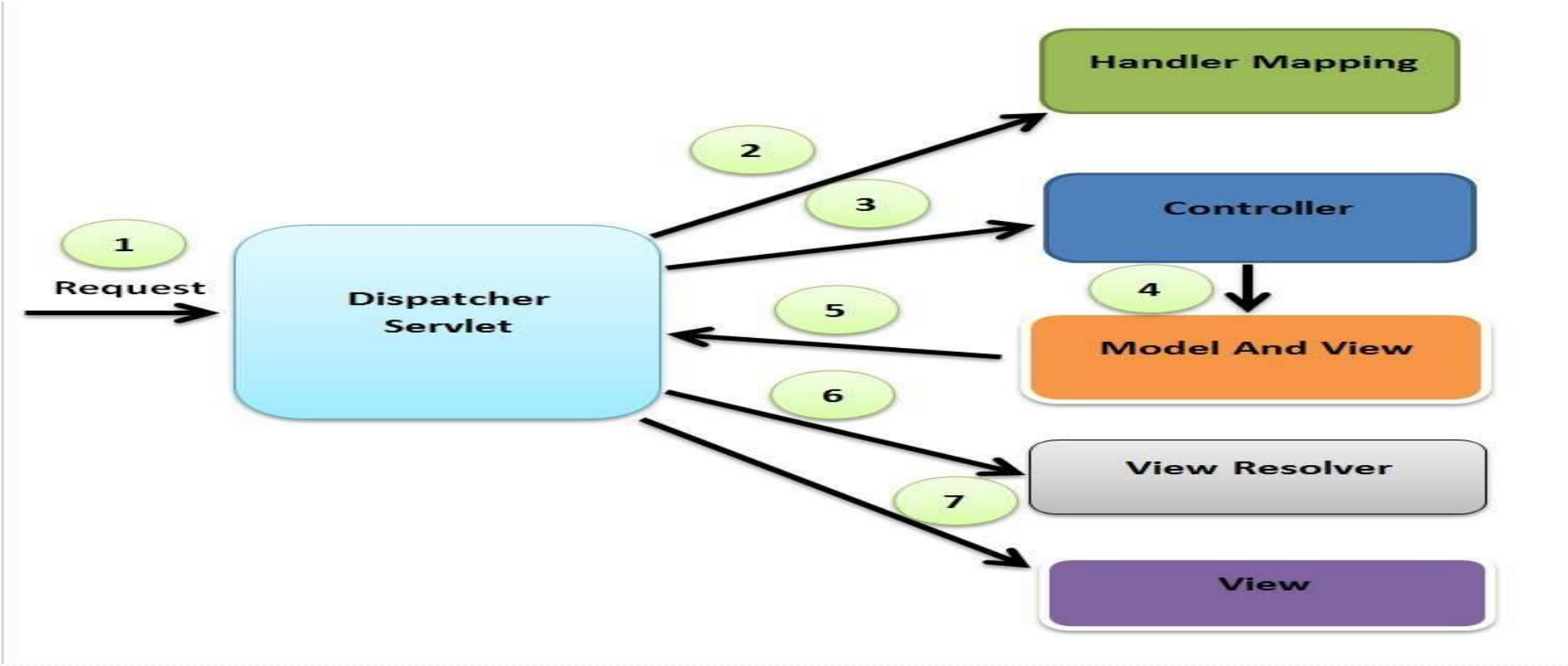
- It gets its name from the fact that it dispatches the request to many different components, each an abstraction of the processing pipeline

1. Discover the request's Locale; expose for later usage.
2. Locate which request handler is responsible for this request (e.g., a Controller).
3. Locate any request interceptors for this request. Interceptors are like filters, but customized for Spring MVC.
4. Invoke the Controller.
5. Call postHandle() methods on any interceptors.
6. If there is any exception, handle it with a HandlerExceptionResolver.
7. If no exceptions were thrown, and the Controller returned a ModelAndView, then render the view. When rendering the view, first resolve the view name to a View instance.



Spring 3.0

- the `DispatcherServlet` is an expression of the “Front Controller” design pattern
- the `@Controller` mechanism also allows you to create RESTful Web sites and applications,
- through the `@PathVarariable` annotation



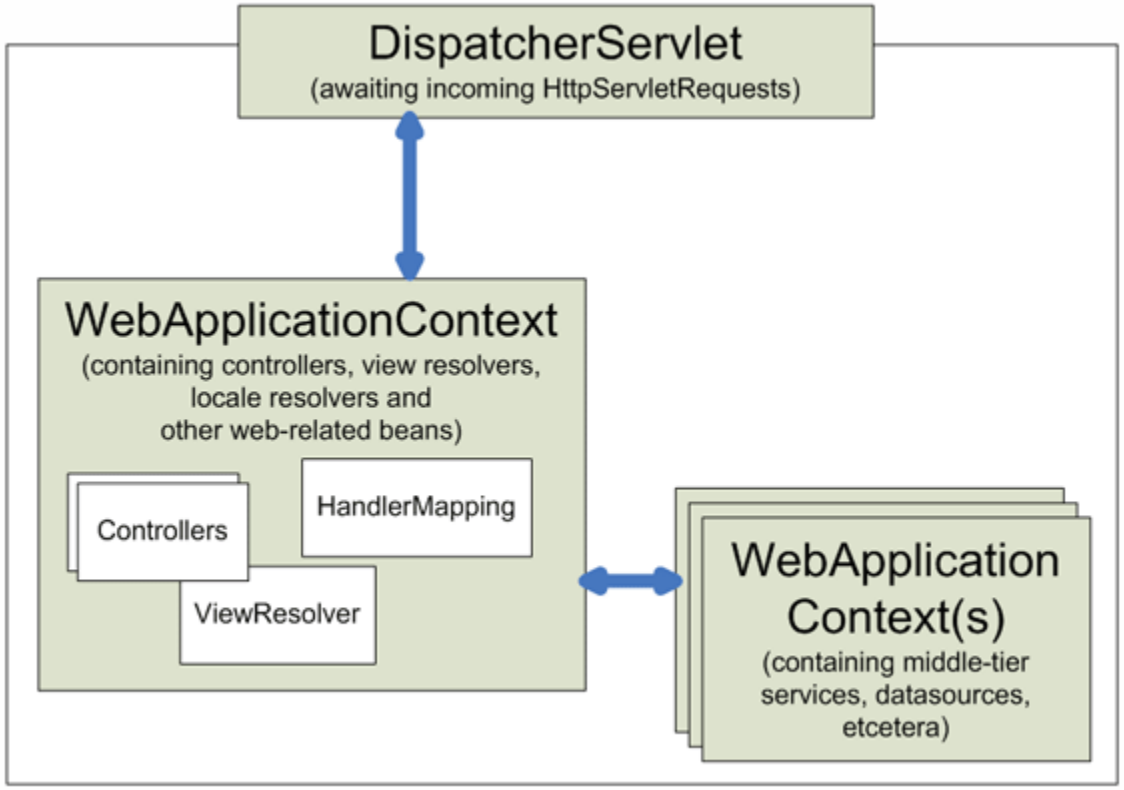
https://docs.spring.io/spring-framework/docs/3.0.0.M4/spring-framework-reference/html/ch15.html

```
<bean id="viewResolver"
  class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/jsp/" />
  <property name="suffix" value=".jsp"/>
</bean>
```

```
public class HomeController extends AbstractController {

    private static final int FIVE_MINUTES = 5*60;
    private FlightService flights;

    public HomeController() {
        setSupportedMethods(new String[]{METHOD_GET});
        setCacheSeconds(FIVE_MINUTES);
    }

    public void setFlightService(FlightService flightService) {
        this.flights = flightService;
    }

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req,
            HttpServletResponse res) throws Exception {
        ModelAndView mav = new ModelAndView("home");
        mav.addObject("specials", flights.getSpecialDeals());
        return mav;
    }

}
```

ThymeleafViewResolver implements the *ViewResolver* interface and is used to determine which Thymeleaf views to render, given a view name

- ApplicationContext
  - WebApplicationContext
    - Contains ServletContext

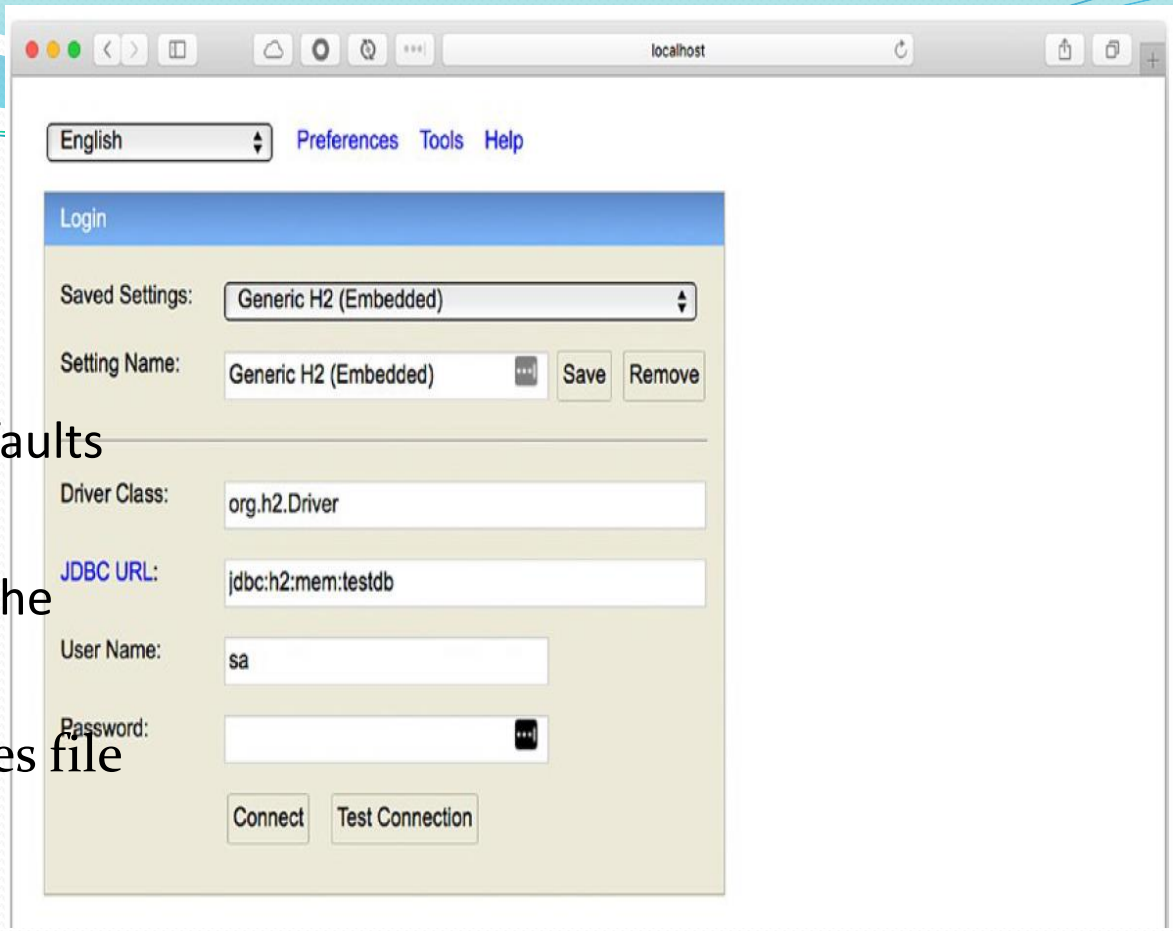


Spring Data Project

- In JDBC, programmers had to download the correct drivers and connection strings, open and close connections, SQL statements, result sets, and transactions, and convert from result sets to objects
- ORM (object-relational mapping) frameworks started to emerge to manage these tasks
  - Hibernate
- They allowed you to identify the domain classes and create XML that was related to the database's tables.
- Spring utilizes such frameworks by following the *template design pattern*.
- It allowed you create an abstract class that defined ways to execute the methods
  - It also created the database abstractions that allowed you to focus only on your business logic.
- It left all the hard lifting to the Spring Framework, including handling connections (open, close, and pooling), transactions, and the way you interact with the frameworks.
- The Spring Data project is the umbrella for several additional libraries and data frameworks, which makes it easy to use data access technologies for relational and non-relation databases (a.k.a. NoSQL).

Connecting to H2

- Spring Boot uses auto-configuration to set sensible defaults when it finds out that your application has a JDBC JAR
- Spring Boot auto-configures the datasource based on the SQL driver in your classpath
- No need to mention any data source in the properties file
- ***To enable H2 console***
  - ***src/main/resources/application.properties***
  - spring.h2.console.enabled=true
- *http://localhost:8080/h2-console*



Felipe Gutierrez, Pro Spring Boot 2, second edition, chapter 5: Data Access with Spring Boot

Connecting MySql

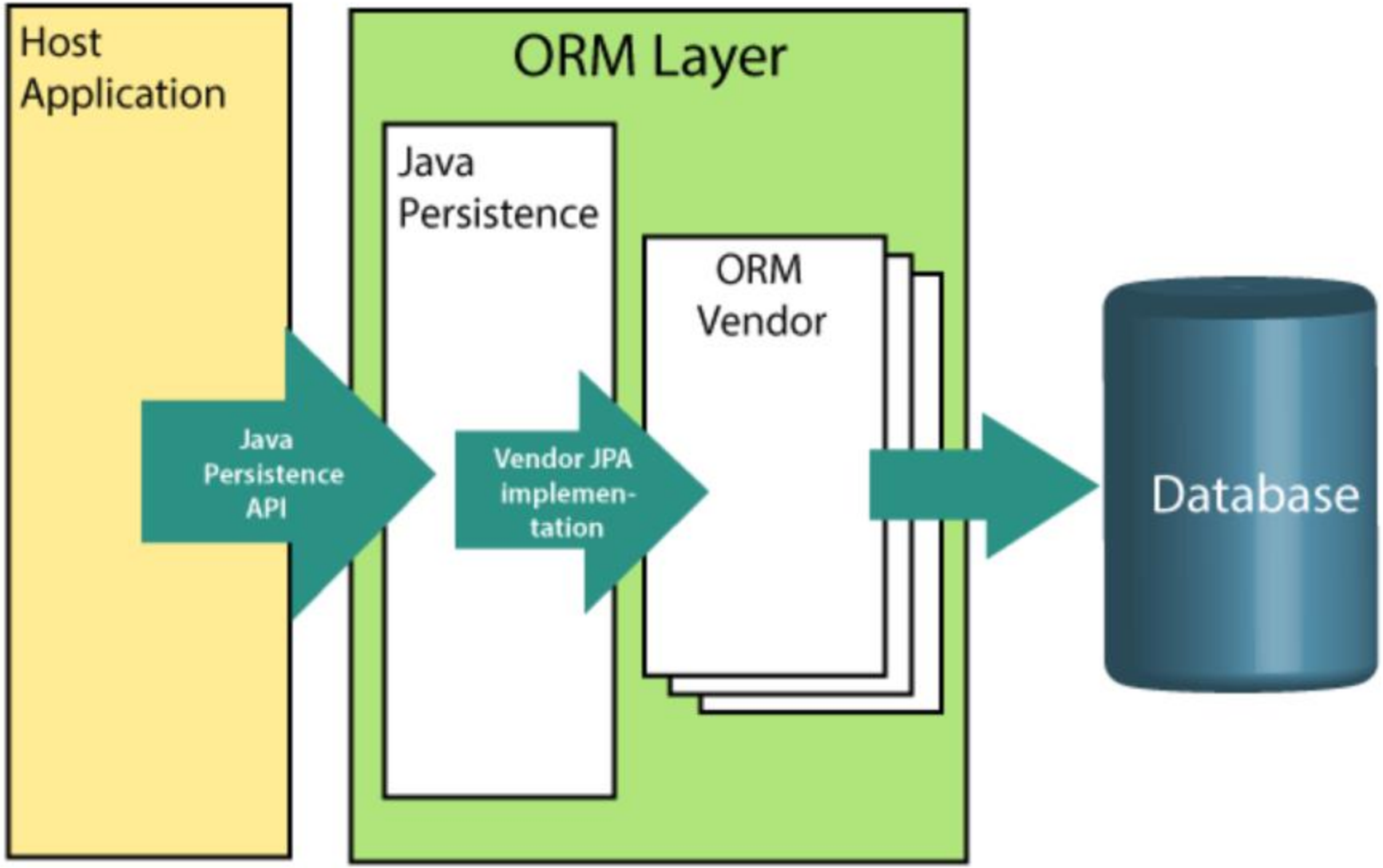
- spring.datasource.url=jdbc:mysql://localhost:3306/testconnect
- spring.datasource.username=root
- spring.datasource.password=1234
- spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

Spring-JDBC

- Another feature that Spring Boot brings to data apps is that if you have a file named *schema.sql*, *data.sql*, *schema-<platform>.sql*, or *data-<platform>.sql* in the classpath, it initializes your database by executing those script files
- So, If you want to use JDBC in your Spring Boot application, you need to add the spring-boot-starter-jdbc dependency and your SQL driver.
- Spring provides a JDBCTemplate class
- The JdbcTemplate class offers you a lot of possibilities to interact with any database engine
  - You can use NamedParameterJdbcTemplate (a JdbcTemplate wrapper) to provide named parameters (:parameterName), instead of the traditional JDBC "?" placeholders.
  - It provides wrapper for ResultSet to access each row and many more

Spring Data JPA

- The JPA (Java Persistence API) provides a POJO persistence model for object-relational mapping
- It follows the *aggregate root* concept
  - Support for repositories (a concept from *Domain-Driven Design*)
- Implementing data access can be a hassle because we need to deal with connections, sessions, exception handling, and more, even for simple CRUD operations
- That's why the Spring Data JPA provides an additional level of functionality: creating repository implementations directly from interfaces and using conventions to generate queries from method names
- Pagination, sort, dynamic query execution support.
- Support for @Query annotations
- JavaConfig based repository configuration by using the @EnableJpaRepositories annotation.


Host
Application
ORM Layer
Java
Persistence
ORM
Vendor
Java
Persistence
API
Vendor JPA
implemen-
tation
Database

Repository Concept

- The most compelling feature of Spring JPA is the ability to create repository implementations automatically, at runtime, from a repository interface.
- We only need to create an interface that extends from a Repository<T,ID>, CrudRepository<T,ID>, or JpaRepository<T,ID>
- The JpaRepository interface offers not only what the CrudRepository does, but also extends from the PagingAndSortingRepository interface that provides extra functionality

the T means the entity (your domain model class) and the ID, the primary key that needs to implement Serializable.

```
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
      <S extends T> S save(S entity);
      <S extends T> Iterable<S> saveAll(Iterable<S> entities);
      Optional<T> findById(ID id);
      boolean existsById(ID id);
      Iterable<T> findAll();
      Iterable<T> findAllById(Iterable<ID> ids);
      long count();
      void deleteById(ID id);
      void delete(T entity);
      void deleteAll(Iterable<? extends T> entities);
      void deleteAll();
}
```

Java Persistence API

In a simple Spring app, you are required to use the @EnableJpaRepositories annotation that triggers the extra configuration that is applied in the life cycle of the repositories defined within of your application

The class that should persist in the database

```
@Entity
public class Person {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private long id;

   private String firstName;
   private String lastName;
Getter and setter methods, constructors both versions }
```

Application properties

Spring Boot provides properties that allow you to override defaults when using the Spring Data JPA. One of them is the ability to create the DDL (data definition language), which is turned off by default, but you can enable it to do reverse engineering from your domain model.

- Hibernate ddl auto (create, create-drop, update)

```
spring.jpa.hibernate.ddl-auto=update
```

- none: The default for MySQL. No change is made to the database structure.
- update: Hibernate changes the database according to the given entity structures.
- create: Creates the database every time but does not drop it on close.
- create-drop: Creates the database and drops it when SessionFactory closes

- `spring.jpa.show-sql=true`
- The simplest way is to dump the queries to standard out

Create the repository

```
public interface PersonRepository extends CrudRepository<Person, Long> {
  List<Person> findByFirstName(String firstName);
}
```

- Invoke the database through dependency injection
- @RepositoryRestResource(collectionResourceRel = "people", path = "people")
  - At runtime, Spring Data REST automatically creates an implementation of this interface.
  - collectionResourceRel -The rel value to use when generating links to the collection resource.
  - Path-The path segment under which this resource is to be exported.

- Then it uses the @RepositoryRestResource annotation to direct Spring MVC to create RESTful endpoints at /people

Application.properties

- `spring.datasource.url=jdbc:mysql://localhost:3306/testconnect`
- `spring.datasource.username=root`
- `spring.datasource.password=1234`
- `spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver`
- `spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect`

  - Advanced translation of PersistenceExceptions to Spring DataAccessException
  - Applying specific transaction semantics such as custom isolation level or transaction timeout

  - Spring Data Jdbc