Spring Data Overview

Spring Data Overview

Spring Data Query Methods

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Spring Data capabilities and features
- Spring Data repositories
- The Repository interfaces
- Defining the JPA entity
- Persisting entities using Spring Data JPA
- Bootstrapping the Spring Data application

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Overview of Data Access Support

- Spring offered several libraries to ease implementation of the data access layer
  - Template classes encapsulate common (and often tedious) technology-specific tasks

```
public Integer getCustomerID(Integer reservationID) {
   String sql = "SELECT CUSTOMER_ID FROM RESERVATIONS WHERE RESERVATION_ID=?";
   return jdbcTemplate.queryForObject(sql, new Object[]{reservationID}, Integer.class);
}
```

  - Support classes define base classes for DAO

```
public class ReservationDAO extends JdbcDaoSupport {
   public int countReservations() {
      String sql = "SELECT COUNT(*) FROM RESERVATIONS";
      return getJdbcTemplate().queryForObject(sql, Integer.class);
   }
}
```

- Templates helped with many of the connection management issues
  - Still required writing native queries and operations

Spring offers many different helpful libraries, frameworks and classes for easing the implementation of your data access layer. Spring offers base class implementations for your DAOs. Depending on the persistent technology chosen, you choose a different base class. Each specific DAO Support class is initialized using the IoC with specific information. For example, the JdbcDaoSupport is initialized from a DataSource, the HibernateDaoSupport from a Hibernate SessionFactory.

One of the important tasks of the DAO Support classes is to provide you with a *Template* class for your persistence technology. These template classes encapsulate many common tasks that are often considered tedious. For example, the JdbcTemplate can be used to fire off a SQL statement without handling the exception, managing the resources, or even the result (for example, when the query is SELECT PRICE FROM SOMETABLE, the price value can be obtained in a single statement).

For each persistence technology, there are various helper classes and small mini frameworks to make things easier for you. For example, when using plain JDBC as the persistent technology, you can use Spring to map the result to a Map or use a custom RowCallbackHandler to map each row from a resultset to an object.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Data Overview

- Spring JPA provided a layer of abstraction relative to relational database queries and operations
- Spring Data provides a layer of abstraction between the application and the entire data layer
  - Including JPA and templates
  - Defines consistent model for Big Data, NoSQL, and Relational datastores
  - Still provides access to specific data store features

While Spring JPA already provided an abstraction layer relative to the relational database queries and operations that need to occur to manage the state of the entities, Spring Data takes the level of abstraction one step further. It defines a consistent model for persisting data for Relational datastores, Big Data and NoSQL data solutions.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Data Capabilities

- Supports relational datastores
  - Through JPA, JDBC and JDBC extensions
- Supports many NoSQL databases
- Provides query capabilities
  - Including a type safe DSL (domain-specific language)
- Supports Big Data such as Hadoop and Splunk
- (Optionally) Provides access through REST APIs
- Uses templates to get to database-specific capabilities
- Repository model supports CRUD operations across data repositories

Spring Data is not just an abstraction layer to our datastore, it provides a consistent approach to storing, retrieving and searching for data. It provides support for SQL and NoSQL databases and even Big Data.

Spring Data even provides solutions for exposing the repository through a REST API.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Data Repositories

- Repositories mediate between domain and data mapping layers
  - Collection-oriented interface working with domain objects
- Basic steps in defining repository
  - Map domain entities (POJOs) to data store
  - Define repository interface, extending base interface
  - Extend repository interface, adding finder methods using various criteria
  - Configure scan for repository
  - Inject repository into application and service layer
- Spring Data handles details of building a repository

The concept of repositories was introduced by Martin Fowler. They mediate between the domain and data mapping layers of applications, providing a collection-oriented interface to working with domain objects.

Every time the state of an entity needs to be persisted in a store, the same steps have to be taken:

1. Mapping the properties of the entity to the store. In a relational database, this would involve mapping the properties to columns in one or more database tables
2. Define a repository interface by extending one of the Spring Data interfaces.
3. Extend the interface by defining additional finder and update functionality
4. Configure the ApplicationContext and make sure the repository is discovered by the Spring runtime
5. Inject the repository (interface) into the applications and service layer.

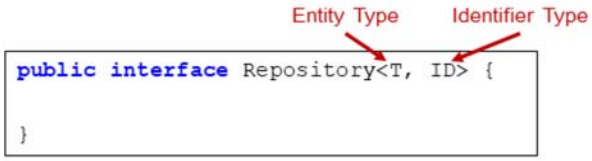
The implementation of the interface is created by Spring Data. We do not have to provide a class that implements the repository interface.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

`Repository` Interface

- Core interface of Spring Data repository
  - Requires domain class and type of id



- Marker interface capturing domain and ID type of entity
  - Base interface for variety of repository interfaces

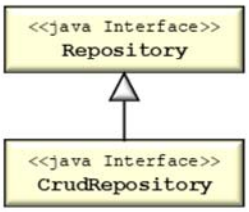
Spring Data defines the Repository interface. This marker interface does not define any methods, but does define the generic types for both the type of the entity and the type of the identifier used to uniquely identify a single instance of the entity.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

CrudRepository

- Defines CRUD operations for managed entity
  - Save (persist) the entity
  - Return entity by identifier
  - Return all entities
  - Return number of entities in repository
  - Delete entity
  - Check existence of entity by providing identifier




```
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();
}
```

The CrudRepository interface extends the Repository interface and defines the method most commonly used to perform the CRUD operations on entity state. It defines the methods to store and remove the state from the datastore and finder methods to find either all instances or an entity by its ID.

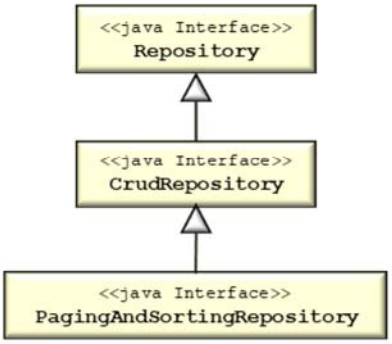
Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

PagingAndSortingRepository

- Extends `CrudRepository`
  - Adds methods for pagination and sorting

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



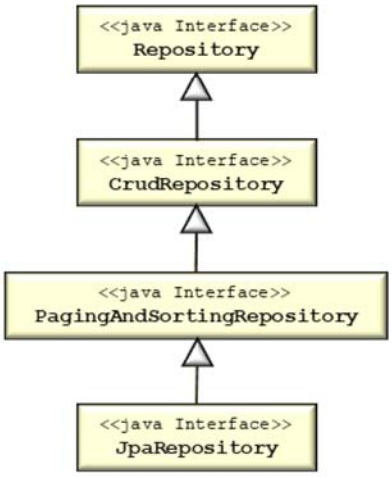
PagingAndSortingRepository adds methods to the CrudRepository making it possible to define both a sorting order and pagination options to finder methods.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

JpaRepository

- Defines JPA extension, adding JPA specific methods
  - List all entities
  - Batch save operation
  - Flush pending tasks
  - Batch delete operations
  - ...




```
public interface JpaRepository<T, ID>
         extends PagingAndSortingRepository<T, ID>,
                 QueryByExampleExecutor<T> {
   List<T> findAll();
   List<T> findAllById(Iterable<ID> ids);
   <S extends T> List<S> saveAll(Iterable<S> entities);
   void flush();
   <S extends T> S saveAndFlush(S entity);
   void deleteInBatch(Iterable<T> entities);
   void deleteAllInBatch();
   T getOne(ID id); //throws EntityNotFoundException
                    //if no entity exists for id
   ...
}
```

JpaRepository extends the PagingAndSortingRepository interface to add several methods that are commonly used in repositories that are based on the Java Persistence API.

One potential downside of using these base repositories is that they expose a large set of operations. There is no easy way to limit the exposure of these methods, such as setting up a read only repository. It is possible to define and implement your own type of repository.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Data JPA

- Repository interface for JPA extends `JpaRepository`

```
public interface ReservationRepository
                 extends JpaRepository<Reservation, Integer>{
}
```

Entity type IdentifierType

- Provides most common CRUD methods

```
@Autowired
private ReservationRepository repository;

public void sample(Reservation reservation, Integer id) {
   repository.save(reservation);
   repository.delete(reservation);
   List<Reservation> reservations = repository.findAll();
   Reservation one = repository.getOne(id);
}
```

To define a DAO interface for a JPA implementation using Spring Data, the interface should extend the JpaRepository interface. The JpaRepository interface generic types define the Entity type and the primary key type of the entity.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining the Entity

- Class is declared as entity using `@Entity`
- `@Table` can be used to define the table name and schema (optional)
- `@Id` defines the identity column
- `@GeneratedValue` defines how the identity will be generated (optional)
- `@Column` can declare
  - Column name
    - When different from property
  - Specific type mapping
  - Columns sizes
  - Nullability, Uniqueness …

```
@Entity
@Table(name = "RESERVATIONS")
public class Reservation {
   @Id
   @GeneratedValue
   private Integer id;
   @Column(name = "NAME")
   private String nameOnReservation;
   private LocalDate arrivalDate;
   private int numberOfNights;
   @Enumerated(EnumType.STRING)
   private Status status;
}
```

To mark a class as an entity whose state is to be managed by JPA, we need to annotate the class with the @Entity annotation.

In addition to this, the @Table annotation can be used to provide information about the table in which the state of this entity must be stored. By default, Hibernate will assume that the name of the database table is identical to the unqualified class name of the class. Since, in our example, the table name is different, we have to add this annotation to define the alternative table name.

It is recommended that all entities have a primary key field that uniquely identifies an instance of the entity. The identifier field within the entity must be annotated using the @Id annotation. When this primary key value is not assigned by the application but must be generated during the insert of the entities state into the database, we must annotate the property with the @GeneratedValue annotation

By default, all properties of a class are considered to be persistent fields. By default, Hibernate will assume that the column names in the database are identical to the property names in the class. When this is not the case or when you want to define additional information about the column, you must use the @Column annotation

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The `Persistable` Interface

- (Optional) Interface for implementation of entities
  - Adds identity (ID) to entity
  - Assists Spring Data in determining if instance is new or has already been persisted
    - Determine whether it should be inserted or updated

```
public interface Persistable<ID> {
   ID getId();
   boolean isNew();
}
```

The Persistable interface is an optional interface that can be implemented by entities that need to be persisted using Spring Data. It requires the definition of an getId method and the isNew method The isNew method is used by the framework to determine if the data from the entity should be added to the datastore or if existing data has to be updated

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

AbstractPersistable

- Abstract base class for definition of entities
  - Generic type defines type of identifier
  - Defines auto generation of primary key
  - Implements `equals` and `hashCode` based on identifier
  - Implements `isNew` based on presence of Identifier

```
public abstract class AbstractPersistable<PK extends Serializable>
                                    implements Persistable<PK> {
   @Id @GeneratedValue private @Nullable PK id;
   public PK getId() {
      return id;
   }
   protected void setId(@Nullable PK id) {
      this.id = id;
   }
   …
}
```

```
@Entity
public class Reservation extends AbstractPersistable<Integer> {
   private Integer reservationNumber;
   @Column(name = "NAME")
   private String nameOnReservation;
   private LocalDate arrivalDate;
   private int numberOfNights;
}
```

AbstractPersistable is an abstract base class that can be used for the definition of entities. It defines an (generic) identifier field, which has already been annotated as the identifier field. This is also specifying that its value will be generated by the framework when the entity is first persisted. It also defines an equals and hashcode implementation based on the identifier of the instance.

The isNew method has been implemented to check the presence of the identifier. When the identifier is null, the entity is assumed new, resulting in the data to be added to the datastore. A non-null identifier causes the data identified by its value to be updated.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Persisting Entities in Spring Data JPA

- Persisting entities can be accomplished using the `save()` method of `CrudRepository` interface
  - Invokes `EntityManager` (JPA) to `persist()` or `merge()`
- Spring Data JPA decides if entity is new or needs to be merged
  - By default, it checks the incoming entity's identity property
    - When null, entity is assumed new and entity is persisted
    - When not null, a merge is attempted
  - When entity implements `Persistable` the `isNew()` method is used instead
- `EntityInformation` class can be overridden to provide alternative approach
  - Requires subclassing the `JpaRespositoryFactory`
    - This approach is not used very often

Once the interface has been defined, its save method can be used to persist the state of the entity in the datastore. Spring Data will rely on the JPA EntityManager to perform the actual insert or update of the data.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Setup Spring Data

- Spring Data requires several classpath dependencies
  - Spring Data Libraries

```
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-jpa</artifactId>
</dependency>
```

  - Persistence Provider

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
</dependency>
```

  - Database Provider

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
</dependency>
```

Just like any other Java Enterprise project, several libraries need to be added to the classpath before you can start developing the application. To develop a JPA persistence layer using Spring Data, we need to add the spring-data-jpa library and define the persistence provider implementation. The examples and exercises in this training use the Hibernate persistence provider.

A database driver also needs to be made available to the application to allow the persistence provider to make a connection to the data store.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

'Enabling' Spring Data

- Transaction management and JPA repositories have to be enabled in context
  - By adding annotations to configuration class

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories
public class JavaConfig { … }
```

- Transaction manager needs to be defined

```
@Bean
public JpaTransactionManager transactionManager() {
   JpaTransactionManager txManager = new JpaTransactionManager();
   txManager.setEntityManagerFactory(entityManagerFactory().getObject());
   return txManager;
}
```

Once all libraries are added to the classpath, a transaction provider needs to be made available to the ApplicationContext in which the persistence provider resides.

To enable the use of JPA repositories and container managed transactions, the configuration class has to be annotated.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

'Enabling' Spring Data (cont'd)

- `EntityManagerFactory` has to be configured
  - Providing Persistence Provider

```
@Bean
public LocalEntityManagerFactoryBean entityManagerFactory() {
   LocalEntityManagerFactoryBean lfb = new LocalEntityManagerFactoryBean();
   lfb.setPersistenceUnitName("jtravel-data");
   lfb.setPersistenceProvider(new HibernatePersistenceProvider());
   return lfb;
}
```

  - Referencing persistence unit
    - Defined by persistence.xml

```
<persistence version="2.0" …>
  <persistence-unit name="jtravel-data">
    <class>com.jtravel.domain.Reservation</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      …
    </properties>
  </persistence-unit>
</persistence>
```

The persistence manager that is used by Spring Data JPA needs to be configured within the application context. In early versions of JPA, the actual configuration of the persistence context was done in an XML file called persistence.xml. When configuring the EntityManagerFactory, it references the name of the persistence-unit defined in the XML file.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

No More XML

- Since Spring 3.1 all configuration can be done in Java
  - The persistence.xml is no longer needed
- `LocalContainerEntityManagerFactoryBean` is used to configure entire context
  - Referencing `DataSource` to use
  - Defining packages that need to be scanned for entities

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
   LocalContainerEntityManagerFactoryBean l =
            new LocalContainerEntityManagerFactoryBean();
   l.setDataSource(dataSource());
   l.setPersistenceUnitName("jtravel-data");
   l.setPersistenceProviderClass(HibernatePersistenceProvider.class);
   l.setPackagesToScan("com.jtravel.domain");
   return l;
}
```

Starting with Spring 3.1, the entire persistence context can be configured in Java, making the persistence.xml obsolete. Using an instance of `LocalContainerEntityManagerFactoryBean`, the datasource and persistence provider that is to be used can all be configured in Java

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Bootstrapping Spring Data (Spring Boot)

- Starter dependency is used to enable Spring Data JPA in Spring Boot
  - Configures Hibernate as default JPA provider
    - No need to configure `EntityManager`

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

- Database dependency has to be added

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
</dependency>
```

- When Spring Data REST is required additional starter is needed

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

With the introduction of Spring Boot, a lot of the configuration has become optional. By relying on auto-configuration and smart defaults, Spring Boot is able to configure a DataSource, transaction manager and entity manager as soon as the libraries are added to the classpath. The EntityManager only needs to be defined when modifications are needed.

Adding the Spring starter dependencies to the build file adds all the library dependencies to the classpath that are needed to build and run the application. When a REST interface is also needed as interface to our repository, the Spring Data REST starter has to be added to the build file, while auto-configuration takes care of the rest.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Boot Embedded Database

- Spring Boot can auto-configure embedded (in-memory) databases
  - Supporting H2, HSQL and Derby
- No database information needs to be provided
  - Type depends on database dependencies found on classpath
- Database needs to be populated on application start
  - SQL scripts can be used to accomplish this
    - `schema.sql` containing database structure
    - `data.sql` containing data to be inserted into the tables
  - Hibernate DDL should be disabled in application.properties

```
spring.jpa.hibernate.ddl-auto=none
```

When a database library is added to the classpath, Spring Boot will automatically configure an in-memory datasource.

To populate the database, a file called schema.sql can be added to the classpath which defines the ddl of the database. A file called data.sql can be added to define the data that is to be inserted into the database table after the application has started.

Spring Boot processes the schema-${platform}.sql and data-${platform}.sql files (if present), where platform is the value of spring.datasource.platform (the database name)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Modifying The Spring Boot `DataSource`

- Spring Boot also auto-configures the `DataSource` bean
  - Depending on database used
- DataSource configuration can be modified
  - Using properties in `application.properties`

```
spring.datasource.url=jdbc:derby://localhost:50505/SpringData
spring.datasource.username=user
spring.datasource.password=password
spring.datasource.driver-class-name=org.apache.derby.jdbc.ClientDriver
```

When the default datasource is not sufficient to support the application, you can configure your own DataSource instance in one of the configuration classes of the ApplicationContext. You can use the Spring Boot properties to define alternative values that are to be used to create the connection to the database.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining a DataSource

- A Custom DataSource can be defined in configuration
  - Defining @Bean annotated method
- DataSourceBuilder assists in building DataSource

```
@Bean
public DataSource dataSource() {
   return DataSourceBuilder.create()
            .driverClassName("org.apache.derby.jdbc.ClientDriver")
            .url("jdbc:derby://localhost:50505/SpringData")
            .username("sa")
            .password("password")
            .build();
}
```

- Spring Boot uses DataSource anywhere one is needed
  - This includes the database initialization process

Instead of using the Spring Boot properties to 'override' the connection details, you can always configure an instance of DataSource in one of your Java configuration files. An instance of DataSourceBuilder can assist in the definition and creation of a DataSource.

Once configured, this datasource will be used by Spring Boot anywhere a DataSource is needed, including the database initialization process used during startup when the schema.sql and data.sql files are being executed.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@ConfigurationProperties

- `ConfigurationProperties` externalizes configuration
  - Simplifies access to properties defined in property files
- Defaults to `application.properties` file on classpath

```
travel.datasource.jdbc-url=jdbc:derby://localhost:50505/SpringData
travel.datasource.username=sa
travel.datasource.password=password
travel.datasource.pool-size=25
travel.datasource.driver-class-name=org.apache.derby.jdbc.ClientDriver
```

  - Supports hierarchical properties
    - Prefix can be defined when adding annotation
- Can be used to externalize database configuration

```
@Bean
@ConfigurationProperties("travel.datasource")
public DataSource dataSource() {
   return DataSourceBuilder.create().build();
}
```

The `@ConfigurationProperties annotation can be used to` externalize the configuration used by the DataSourceBuilder. Spring Boot defines a default properties file called, application.properties. As the application grows, the number of properties in this file can grow to an amount that is hard to manage.

@ConfigurationProperties works best with hierarchical properties that all have the same prefix. Within the properties file it is now possible to define hierarchies of properties, where all properties at a certain level have the same prefix

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 8: Spring Data JPA Using Spring Boot

~/StudentWork/code/spring-data-jpa-boot/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 9: Spring Data JPA Using Spring Boot (Part 2)

~/StudentWork/code/spring-data-jpa-boot-db/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 10: Spring Data JPA (Without Spring Boot)

~/StudentWork/code/spring-data-jpa/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Data Query Methods

Spring Data Overview
Spring Data Query Methods

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Querying data using Query methods
- Query builder mechanism
- Handling an Absence of Value
- Pagination and Ordering
- Asynchronous query methods
- Count and Delete Derived Query methods

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Querying Data

- Spring Data JPA defines several mechanisms for querying data
  - Query methods
  - `@Query` annotation
  - Custom repository implementations
- Examples in this lesson define queries on these entities

```
@Entity @Table(name = "RESERVATIONS")
public class Reservation {
   @Id @GeneratedValue(strategy =IDENTITY)
   private Integer id;
   private Integer reservationNumber;
   private String nameOnReservation;
   private LocalDate arrivalDate;
   private int numberOfNights;
   @Enumerated(EnumType.STRING)
   private ReservationStatus status;
   @ManyToOne
   private Hotel hotel;
}
```

```
@Entity @Table(name = "HOTELS")
public class Hotel {
   @Id
   @GeneratedValue(strategy =IDENTITY)
   private Integer hotelId;
   private String name;
   private String city;
   private String state;
   private int numberOfRooms;
}
```

To find data within the datastore, Spring Data provides several mechanisms for defining the query that needs to be executed. This lesson focuses on the definition of query methods in the interface.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Query Methods

- Query methods can be added to interface
  - Database queries will be generated from method names
- Can be used for both selection and modification of data

```
public interface ReservationRepository extends JpaRepository<Reservation, Integer> {
   List<Reservation> findByArrivalDate(LocalDate checkIn);
   Integer countByNameOnReservation(String name);
   List<Reservation> removeByReservationNumber(Integer reservationNumber);
}
```

- No implementation of the interface needs to be developed
  - Implementation will be generated by Spring Data

Besides the methods that are 'inherited' by the JpaRepository, the DAO interface may also define other query methods. When using Spring Data, you will not have to write an implementation of this interface. An implementation of the interface will be generated.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Query Return Types

- Query methods may return zero or more results
  - Exact list of valid return type is store-specific
- A Query method may return
  - void
  - A primitive (or its wrapper)
  - A unique entity
    - Null is returned when entity is not found
    - Exception is thrown when multiple entities are found
  - A reference to a Collection of entities
    - `Iterator<T>`, `Collection<T>`, `List<T>`
  - An `Optional` instance
    - Empty optional is returned when entity is not found
    - Exception is thrown when multiple entities are found
  - A Java 8 `Stream<T>` instance

When searching for data, the method may return zero or more instances of the entity. While some methods are expected to return zero or more instances, methods can also be defined to return zero or one value. Keep in mind that when defining a method to return at most one instance, if more than one instance is found, an IncorrectResultSizeDataAccessException runtime exception will be thrown.

The return types shown above lists return types generally supported by Spring Data repositories. However, store-specific implementations might allow for additional return types or might not even support all the types listed above.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Handling an Absence of Value (Null)

- Querying for data might result in zero or more results
- When no results are found method may return
  - If returning a collection, an empty collection
  - An Optional (Java 8)
  - Variety of wrappers from Google, Scala, and other frameworks
  - A 'simple' null value
- When returning `Collection`, wrapper or `Stream`, result will never be null

What is returned when no results are found as a result of executing the query depends on the return type of the of the query method. When the return type is a Collection type an empty collection is returned.

Methods that are expected to return zero or at most one entity may return a ‘simple’ null value, but can also be defined using the Optional class introduced in Java 8.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Nullability Annotations

- Nullability constraints can be defined on methods
  - `@org.springframework.lang.NonNull`
    - Used at parameter or return value
  - `@org.springframework.lang.Nullable`
    - Used at parameter or return value
- Checking of nullability constraints can be activated on package level
  - Adding `@org.springframework.lang.NonNullApi` to `package-info.java`

```
@org.springframework.lang.NonNullApi
package com.jtravel.domain.repository;
```

Checking of nullability constrains can be enabled at package level, by annotating the package-info.java file with the @NonNullAPI annotation.

Once nullability constraints are enabled invocations of query methods are validated at runtime. When query executions violate the constraint, exceptions are thrown.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Nullability Constraints

- When checking of nullability constraints is enabled
  - `EmptyResultDataAccessException` is thrown when query does not produce result
  - `IllegalArgumentException` is thrown when method parameter is null
  - To allow individual methods to return null, `@Nullable` must be added

```
public interface NullableRepository
                        extends JpaRepository<Reservation, Integer> {
   List<Reservation> findByNameOnReservation(@NonNull String name);
   @Nullable
   Reservation findByReservationNumber(Integer reservationNumber);
}
```

- `@NonNull` is not needed when `@NonNullApi` applies
  - Is default behavior

When a method returns null while it is declared as non-nullable, an `EmptyResultDataAccessException is thrown`. When null is a valid return value for a single method (or a valid value for a field), you will have to opt-in the nullable value by adding the @Nullable annotation.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Query Builder Mechanism

- Query builder strips prefix from method name
  - find…By, read…By, query…By, count…By, get…By
- Limiting expressions can be added to 'Select' clause
  - Limiting result to one instance

```
Reservation findOneByReservationNumber(Integer reservationNumber);
```

  - Using 'Distinct'

```
List<Reservation> findDistinctByNameOnReservation(String name);
List<Reservation> findDistinctReservationByNameOnReservation(String name);
```

- First 'By' in method name indicates start of criteria

Query methods can be defined in the repository interface. The actual query that is executed is built by parsing the name of the method.

When parsing the query method, the part of the name before the first 'By' is considered the 'selection' prefix. It can be used to limit the number of results by adding keywords like 'One' or 'Distinct' , after the first 'By' in the method name the selection criteria can be specified.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Limiting Query Results

- Number of results can be limited
  - Using `first` or `top` keywords

```
Reservation findFirstByArrivalDate(LocalDate checkIn);
```

- Optionally, the number of results can be added
  - By appending numeric value to first or top keyword

```
List<Reservation> findTop5ByArrivalDate(LocalDate checkIn);
```

  - When no value is specified, 1 is assumed

To limit the number of entities that is being returned by the query, the first or top keywords can be added before the first 'By' in the statement. Appending a number value to these keywords allows us to define the number of records that is to be returned. When no value is defined, 1 is assumed.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The Selection Criteria

- Selection criteria is defined using properties of managed bean

```
Reservation findOneByReservationNumber(Integer number);
List<Reservation> findByNameOnReservation(String name);
List<Reservation> findByArrivalDate(LocalDate checkIn);
```

```
@Entity
public class Reservation {
   @Id
   private Integer id;
   private Integer reservationNumber;
   private String nameOnReservation;
   private LocalDate arrivalDate;
   @ManyToOne
   private Hotel hotel;
   …
}
```

```
@Entity
public class Hotel {
   @Id
   private Integer hotelId;
   private String name;
   private String zipCode;
   …
}
```

- Nested properties can be referenced

```
List<Reservation> findByHotelName(String name);
List<Reservation> findByHotelZipCode(String zipCode);
```

When defining the selection criteria of the statement, the property expressions can reference properties defined within the managed entity. In the example shown above, the queries reference properties defined in the Reservation class.

Constraints can also be defined by referencing nested properties of the entity class. The Reservation has a ManyToOne relationship to the Hotel entity. To query all the reservations for a hotel, the query method can be defined as ‘findByHotelName’. Spring Data will first look at the Reservation entity itself to determine if a hotelName property was defined as a direct property of this class. When this is not the case, the algorithm will split the method name at the camel case parts and will try to find the corresponding property by traversing the properties of the Reservation entity (hotel.name)

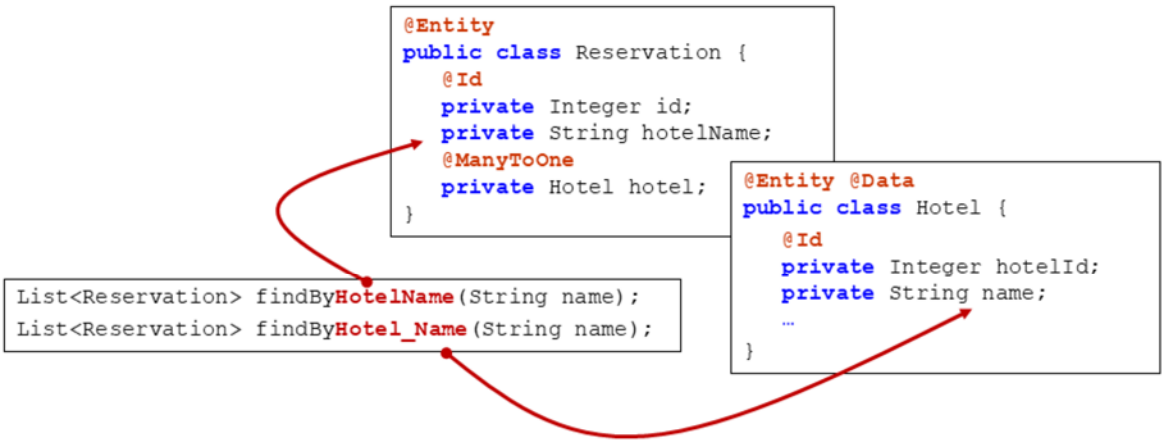
When parsing the properties, Spring Data will always split a property name from right to left. In the case of the ‘findByHotelZipCode’ method. It will split ‘HotelZipCode’ into ‘HotelZip’ and ‘Code’ and will try to locate a property in the reservation class with the name, ‘hotelZip’. When this property is found, it will be used and the remainder of the parameter name to match to nested properties. When the property is not found, it will split at the previous Camelcase (‘Hotel’ and ‘ZipCode’) in an attempt to find the properties.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Nested Property Expressions

- Underscore (_) can be used to explicitly define traversal points
  - Required to resolve ambiguity between properties



Sometimes the property parsing process of Spring Data does not result in finding the correct property. In the example shown above, the Reservation entity contains a hotelName property, while the Hotel entity that is referenced by the hotel property has a ‘name’ property.

When defining a finder method called findByHotelName, Spring Data will query the property of the Reservation entity and NOT the name property defined within the Hotel entity. To resolve ambiguity between properties when defining finder methods, an underscore can be used to explicitly define the traversal points in the parameter.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Repository Query Keywords

- A large number of query keywords have been defined
  - Used to define query conditions in method name

Repository query keywords		
And	Null / IsNull	After / IsAfter
Or	NotNull / IsNotNull	Before / IsBefore
Exists	Not / IsNot	Between / IsBetween
In / IsIn	IsNotEmpty / NotEmpty	Near / IsNear
Is / Equals	Like / IsLike	IsEmpty / Empty
False / IsFalse	NotIn / IsNotIn	Within / IsWithin
True / IsTrue	NotLike / IsNotLike	LessThan / IsLessThan
GreaterThan / IsGreaterThan	LessThanEqual / IsLessThanEqual	Regex / MatchesRegex / Matches
Containing / IsContaining / Contains	EndingWith / IsEndingWith / EndsWith	
StartingWith / IsStartingWith / StartsWith	GreaterThanEqual / IsGreaterThanEqual	

The table shown above provides an overview of all the keywords that can be defined within a finder method to specify more detailed query conditions.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Query Conditions

- Multiple properties can be defined to define condition
  - Concatenated using `And` or `Or` keywords

```
List<Reservation> findByNameOnReservationAndArrivalDate(String name, LocalDate date);
```

- Operators can be added to property expressions
  - `Between`, `LessThan`, `GreaterThan` and `Like`

```
List<Reservation> findByArrivalDateBetween(LocalDate start, LocalDate end);
List<Reservation> findByNameOnReservationLike(String name);
```

  - Supported operators may vary by datastore
- The case of the property value can be ignored
  - setting an `IgnoreCase` flag for individual properties

```
List<Reservation> findByNameOnReservationIgnoreCase(String name);
```

  - Setting `AllIgnoreCase` to ignore case for all properties

```
List<Reservation> findByNameOnReservationOrHotelNameAllIgnoreCase(String name,
                                                                  String hotel);
```

To query entities using multiple properties, the And or Or keywords should be placed within the different property names in the query.

Operators like Between, LessThen and GreaterThan can be used to define the upper or lower bounds of a property value.

When the case of a property value is to be ignored, the IgnoreCase can be added to a single value. When adding AllIgnoreCase to the method name, the case of the values of all properties within the statement is ignored.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining Ordering

- Ordering can be defining by appending `OrderBy` clause
  - Referencing property to be sorted on
  - Defining sorting direction using `Asc` or `Desc`

```
List<Reservation> findByNameOnReservationOrderByArrivalDateAsc(String name);
```

To define the order of the entities in the resultset, the OrderBy keyword can be added to the statement. The sorting direction can be defined by added either ‘Asc’ or ‘Desc’ to the method name.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Pagination

- A `Pageable` parameter is added to specify pagination
  - Dynamically adding paging to static query

```
Page<Reservation> findByArrivalDate(LocalDate date, Pageable pageable);
```

- `PageRequest` defines paging to be used by query
  - Implementation of the `Pageable` interface

```
repository.findByArrivalDate(LocalDate.now(), PageRequest.of(0, 5));
```

Page number Page size

- Method should return `Page<T>`, `Slice<T>` or `List<T>`

When the result set contains a large number of entities, do you really want all those entities in a single collection or would you rather have them in smaller sets of data.

A parameter of type org.springframework.data.domain.Pageable can be defined in the finder method to allow the client to add pagination to the selection process.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Pagination Returning Page

- `Page` object contains sublist of entity instances

```
page.forEach(System.out::println);
Stream<Reservation> stream = page.get();
```

- Knows total number of elements (and pages)

```
long totalElements = page.getTotalElements();
int totalPages = page.getTotalPages();
Pageable nextPage = page.nextPageable();
```

  - Additional 'SELECT COUNT' is performed to acquire this information!

When the return type of the method is of type org.springframework.data.domain.Page, the instance that is being returned contains a sublist of all the entity instances that match the criteria. The instance of Page also knows the total number of elements (and as a result the number of available pages) that can be obtained.

In order for the Page instance to know the total number of entities, an additional Select Count statement must be executed.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Paging Returning `Slice` or `List`

- To avoid the overhead of additional select (count), the `Slice<T>` or `List<T>` return type can be used
  - Slice only knows about previous and next

```
Slice<Reservation> findByArrivalDate(LocalDate date, Pageable pageable);
```

```
Slice<Reservation> slice = repository.findByArrivalDate(today, PageRequest.of(0, 5));
Stream<Reservation> stream = slice.get();
if (slice.hasNext()) {
   Pageable next = slice.nextPageable();  //Page 1
   Slice<Reservation> slice2 = repository.findByArrivalDate(LocalDate.now(), next);
   …
}
```

  - The `hasNext` method should be used to avoid null return value from `nextPageable`

To avoid the overhead of the additional Select Count statement, the return type of the method can be changed into a List or an instance of Slice. While a Slice does not know the total number of entities available, it does allow the client to obtain the next and previous page (when available)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Sorting

- Sorting can also be defined by adding `Sort` parameter

```
List<Reservation> findByNameOnReservation(String name, Sort sort);
```

  - Defining multiple sort properties and sorting direction

```
Sort sort = Sort.by("nameOnReservation").descending()
                .and(Sort.by("arrivalDate")).ascending();
List<Reservation> fetchByArrivalDate =
                        repository.findByNameOnReservation("Fred", sort);
```

- Can also be defined when using Paging

```
Sort sort = Sort.by("nameOnReservation").descending();
PageRequest pageable = PageRequest.of(0, 10, sort);
Page<Reservation> fetchByArrivalDate = repository.findByArrivalDate(today, pageable);
```

```
Page<Reservation> findByArrivalDate(LocalDate date, Pageable pageable);
```

Instead of defining the sorting requirements as part of the method name, an parameter of type Sort can also be added to the finder method. Using the fluent API of the Sort class, detailed sorting requirements can be specified, spanning multiple entity parameters and different sorting directions.

Overloaded methods of PageRequest, allow for the definition of sorting in combination with pagination.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Asynchronous Query Methods

- Query methods can be defined as asynchronous
  - Method must be annotated with `@Async`
  - Asynchronous method execution capability has to be enabled

```
@Configuration @EnableAsync
public class JavaConfig {
   …
}
```

```
@Async
CompletableFuture<Reservation> findOneByNameOnReservation(String name);
```

- `@Async` annotated query methods may return
  - `Future<T>`
  - `CompletableFuture<T>` (introduced in Java 8)
  - org.springframework.util.concurrent.ListenableFuture

When asynchronous method invocations are enabled within the ApplicationContext (configuration class has been annotated with @EnableAsync), finder methods of Spring Data can be annotated with @Async to make sure the execution of these statements is also done asynchronously.

The return types of these asynchronous methods can now be either Future, CompletableFuture (Java 8) or an ListenableFuture type.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Asynchronous Query Methods (cont'd)

- By default, the `SimpleAsyncTaskExecutor` is used to run an asynchronous query
- Executor name can be defined as attribute of `@Async`

```
@Async("queryExecutor")
CompletableFuture<Reservation> findByNameOnReservationAndArrivalDate(
                                    String name, LocalDate arrivalDate);
```

  - Executor must be defined within context

```
@Configuration
@EnableAsync
public class JavaConfig {
   @Bean(name = "queryExecutor")
   public Executor threadPoolTaskExecutor() {
      return Executors.newFixedThreadPool(10);
   }
}
```

By default, an instance of SimpleAsyncTaskExecutor is used to execute the asynchronous task. When you need to be in full control of the executor that is to be used for the execution of these queries, you can define the executor that is to be used in the context. The Executor name (bean name) of the executor to use can be defined as attribute of the @Async annotation.

When @Async is applied at class level, the executor that is defined applied to all methods within the class/interface. When applied at method level, it only applies to that method and overrides the values defined at class level.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Count and Delete Derived Query Methods

- Query derivation can also be used for count and delete queries
- For COUNT queries, method must start with `count` keyword
  - Method should return `long` (or `Long`)

```
long countByArrivalDate(LocalDate date);
```

- For DELETE queries, method must start with `delete` or `remove` keyword
  - Method should return `long` (or `Long`) , `Collection` or `void`

```
List<Reservation> deleteByReservationNumber(Integer reservationNumber);
```

With the introduction of Spring Data v1.7, it became possible to define methods for counting and deleting entities.

Count methods must start with the ‘count’ keyword and should return a long value.

To delete entities, the method should start with either the delete or the remove keyword. These methods can return a long (returning the number of deleted entities), a collection of entities that were removed, or voids.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 11: Spring Data Query Methods

~/StudentWork/code/spring-data-query-methods/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Session: Implementing REST with Spring

REST principles
Introduction to RESTful Services in Spring
Introduction to REST Clients in Spring
Bootstrapping the REST application
Content Representation
Implementing the REST Service

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

REST principles

REST principles
Introduction to RESTful Services in Spring
Introduction to REST Clients in Spring
Bootstrapping the REST application
Content Representation
Implementing the REST Service

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Introduce the six architectural constraints of REST
- Introduce resources and resource representations
- Best practices for defining Resource URIs

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Representational State Transfer

- REST stands for Representational State Transfer
  - Architecture style for designing loosely coupled applications over HTTP
- REST defines 6 architectural constraints
  - Uniform interface
  - Client–server
  - Stateless
  - Cacheable
  - Layered system
  - Code on demand (optional)
- REST does not define low-level implementation styles

To address some of the concerns about the Web's scalability in the early 1993 Roy Fielding (co-founder of the Apache HTTP Server Project) defines the six architectural constraints of the Web's architectural style

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Architectural Constraint: Uniform Interface

- A single resource should have only one logical URI
- All resources within a system should follow the same representation standards
  - Using naming standards across resources
  - Link formats should follow the same standards
  - Data format (XML, JSON, …) should be consistent
  - Common approach should be used to access or modify resource
    - Consistent use of HTTP methods
- Developers should be able to use similar approach across multiple APIs in system

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Architectural Constraint: Client-Server

- REST defines a client-server architecture
  - Server stores and manipulates (a set of) information
  - Server makes information available to client
    - Using efficient data format (e.g. XML, JSON)
  - Client retrieves information from server
  - Client makes subsequent requests to retrieve or modify information
- Client and server must be able to evolve separately
  - Changes to the interface (API) between client and server should be carefully managed and implemented
  - Dependencies between the client-side of the API and the server-side of the API should be limited
    - Client should only be aware of resource URIs

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Architectural Constraint: Stateless

- No client-specific session state should be maintained on the server
  - Each request is handled as new request
  - Each request should be handled in complete isolation
- Client is responsible for managing its state
  - Request should contain all information to service request
    - Contained in URL, request parameters, headers or body
- When resources requires authentication, client needs to re-authenticate for every request
- Being stateless adds several advantages to system
  - Less resources contributes to scalability of server
  - Allows service to be replicated to scale up
    - Multiple requests from client can be handled by different servers (without replicating the state on the server-side)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Architectural Constraint: Cacheable

- Resources may be cached by server or intermediary component(s)
  - Might result in performance improvement for client
  - Reducing the load on the server contributes to scalability
  - Hide network failures
- Form of optimistic replication (lazy replication)
- Cacheable resources should be identifiable
  - Resources MUST declare themselves as cacheable
- HTTP headers can define who and how long a resources may be cached
  - Expires : Defines expiry time of resource
  - Cache-Control : If and how long a resource is cacheable
  - Etag : Uniquely identifies resource state
  - Last-Modified : Last date/time of resource modification

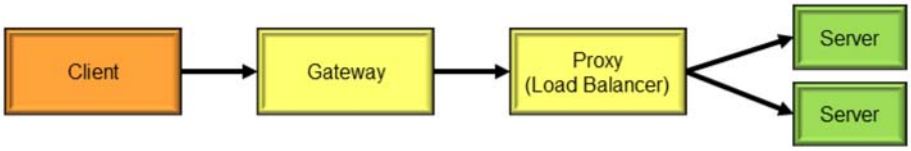
The service does not try to guarantee 100% consistency between itself and its clients unless absolutely critical

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Architectural Constraint: Layered System

- Each layer in the architecture should serve a specific function
- Each layer should be able to act independently
  - Only interacting with immediately adjacent layers
- Several layers might be part of system
  - Proxy layer acting as load balancer
  - Layers responsible for caching
  - Gateways translating request to other protocols
- Layered systems enforce more separation of concerns
  - Minimizing risk of coupling between components



Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Code on Demand (Optional)

- REST optionally provides 'code on demand'
- Response does not have to be only static information
  - Executable code can be returned by resource
    - Scripts, plug-ins, ...
- Not all clients might be able to execute the code
  - REST components cannot rely on logic being executed

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Resources

- All named information can be a resource
  - Documents, images, collections of information, ...
- A Resource defines type, data and relationships to other resources
- A Resource is made accessible through hyperlink
  - Identified by Uniform Resource Identifiers (URIs)
- A Resource defines conceptional mapping of entities
  - A Resource does not change, but available entities might

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Resource Representations

- Resource representation contains data and metadata
- Metadata provides information about representation
  - Resource metadata includes links to related resources
  - Control data provides information about request or response
    - Caching information, last modified information
  - Media type – defines the type of the data in request or response
- The Media type is a key component of a REST architecture

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Hypermedia as Engine of Application State

- Hypermedia: an extension of hypertext
  - Includes images, video, audio, texts and links
- Interactions should be driven entirely by hypermedia
- Client should only require limited knowledge of service
  - Entry point of the service
  - Media type(s) of resource representation(s)
- Link relations should be discoverable for clients

- REST is more than performing operations based on HTTP method
  - GET, POST, PUT, DELETE, HEAD, ...
- JSON resources does not make API RESTful
  - JSON itself does not define concept of links

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Resource Archetypes

- REST defines four distinct resource archetypes
  - Document
  - Collection
  - Store
  - Controller
- API should align each resource with a single archetype

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Archetype Document

- Represents a single object instance within a system
  - For example, a single object instance or a single database record
- Defines base archetype for other archetypes
- Resource defines
  - One or more fields with values
  - Links to other (related) resources
    - May have child resources that represent sub concepts
- Should be referenced using a "singular" noun

```
/reservation/{reservationID}
```

The three other archetypes should be considered specializations of the Document archetype

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Archetype Collection

- Server-managed collection of resources
- New resources can be added to collection
  - Collection responsible for managing individual resources
  - Collection determines URI of each resource contained within
- Should be referenced by plural noun
  - Reflecting (uniform) content of collection

```
/reservations
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Archetype Store

- Defines a client-managed resource repository
  - Allowing client to add, delete and retrieve resources
- Stores not responsible for creating new resources
  - URI or each resource is chosen by the client
    - When resource is added to the store

```
POST  /reservations/{reservationID}
```

- Should be referenced by plural noun
  - Reflecting (uniform) content of the store

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Archetype Controller

- Models a procedural concept
  - Defines specific actions not logically mapped to one of the standard methods (CRUD)
- Similar to executable functions
  - With inputs and return values
- Should be referenced by a verb
  - Describing the action resulting from the request
  - Typically last segment in resource URI

```
/reservations/{reservationID}/resendConfirmation
```

- Often considered bad practice in REST

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Best Practices: Resource Naming

- Use of lowercase letters in URI is preferred
  - URI's are defined as case-sensitive
    - Except for scheme and host component of URI
- Forward slash (/) indicates hierarchical relationship
  - URI should not end with slash
- Hyphens (-) can be used to improve readability
  - Underscores (_) should be avoided
- Do not include function to be performed in URI
  - Use HTTP Method to define operation
- Consistent naming of resources improves readability
- Use 'singular' name to reference singleton resource

```
/reservation/{reservationID}
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Best Practices: Resource Naming (cont'd)

- Use 'plural' name to reference collection resource

```
/reservations
```

  - ...or a singleton within a collection

```
/reservations/{reservationID}
```

- Resources may define 'sub-collections'
  - Referencing a sub-collection

```
/reservations/{reservationID}/drivers
```

  - Referencing a singleton resource inside a sub-domain

```
/reservations/{reservationID}/drivers/{driverNumber}
```

- URI parameters should be used to define filters
  - Collection filters, sorting or pagination

```
/reservations?category=SUV
```

```
/reservations?sort_by=asc(name)
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Introduction to RESTful Services in Spring

REST principles

Introduction to RESTful Services in Spring

Introduction to REST Clients in Spring

Bootstrapping the REST application

Content Representation

Implementing the REST Service

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

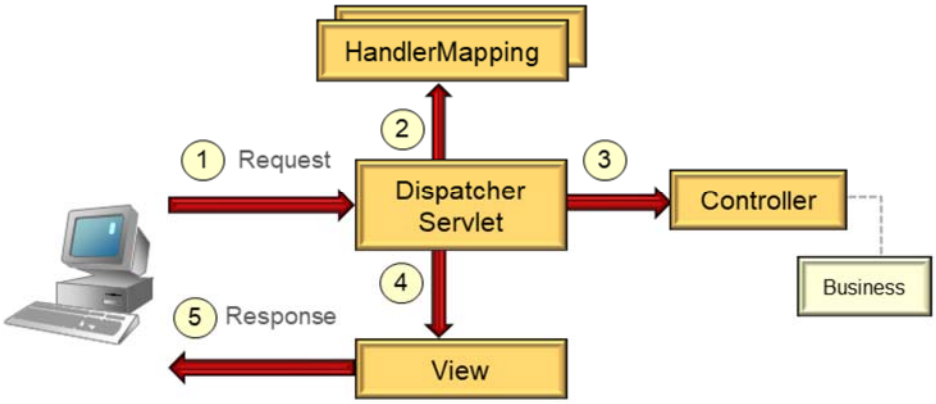
- Discuss the request-response cycle of REST requests
- Defining a REST Controller in Spring
- Explain the `@ResponseBody` annotation
- Define request mappings
- Use path variables

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

REST Applications in Spring

- Originally, REST applications where developed using Spring MVC framework
  - Built on top of existing (configuration heavy) API
- Tried to apply REST paradigm to 'old' model
  - MVC controller relies on View technology



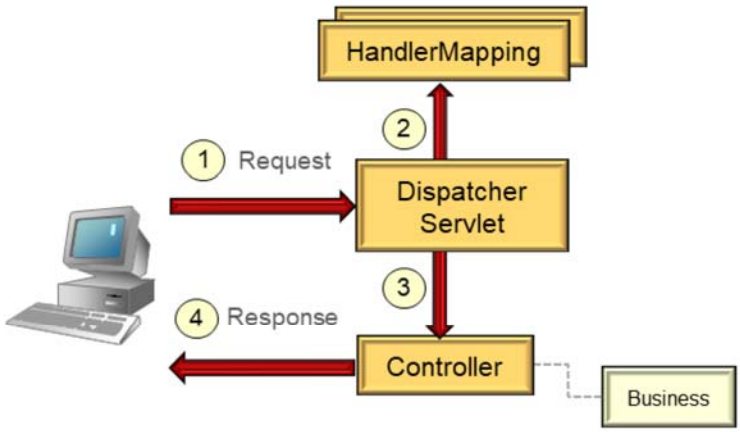
Initially the implementations of Spring REST made use of the already existing Spring MVC framework. Not only does Spring MVC require a lot of boilerplate code and configuration, the REST programming model does not really fit into the MVC model. In Spring MVC the DispatcherServlet relies on a ViewResolver to determine which view should be returned to the client, while in Spring REST the controller returns data directly to the client.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

REST Applications in Spring (cont'd)

- A RESTful controller returns the object data as is
  - Written directly to HTTP response as JSON or XML



- Spring 3.0 introduced more lightweight approach
  - Using HTTP message converters and annotations

A RESTful controller returns the outcome of the controller invocation directly to the client that made the request. The object instance that is returned by the controller will be 'transformed' into JSON or XML, but the ViewResolver is not needed to determine which view should be returned.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining the REST Controller

- Controller implementations should be annotated with `@Controller`
  - Specialization of `@Component` annotation
- Used together with `@RequestMapping` annotation(s)
  - Mapping requests to class (and its methods)

```
@Controller
@RequestMapping("/hello")
class HelloController {
   @RequestMapping("/welcome")
   @ResponseBody
   public String sayHello() {
      ...
   }
}
```

- Class and methods do not need to be defined as public
  - Receives HTTP requests from the Spring front controller

To define a REST controller in Spring, the class should be annotated with the @Controller annotation. This annotation is a specialization of the @Component annotation, which allows it to be automatically discovered by the Spring context when classpath scanning is enabled.

The RequestMapping annotation can be added to the class and its methods to map request URI to the methods defined within the controller.

The controller class and the controller methods do not have to be defined as public. Since the methods are only invoked by the Spring front controller, the protected or default access modifiers can also be used.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Using the @ResponseBody Annotation

- Spring MVC would 'normally' use result of method to determine View to be displayed
  - ViewResolver could be used to determine View
- Spring REST controllers return data object
  - Should be written directly to HTTP response
- Methods should be annotated with @ResponseBody
  - Enables serialization of method response to http response

```
@RequestMapping("/welcome")
@ResponseBody
public String sayHello() {
   …
}
```

- Could also be defined at class level
  - 'Inherited' by all methods

```
@Controller
@ResponseBody
@RequestMapping("/hello")
class HelloController {   ...   }
```

By default the result returned by a controller method would be used by the framework to determine what view must be presented to the client. When defining REST controllers, the framework must be told to bypass the ViewResolver. To accomplish this the method (or the class) should be annotated with the @ResponseBody annotation.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@RestController

- Spring 4 introduced `@RestController` annotation
  - Convenience annotation, combining `@Controller` and `@ResponseBody`

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController {
    …
}
```

```
@RestController
@RequestMapping("/rental")
public class ReservationController {
    @RequestMapping("/reservations")
    public List<Reservation> getAllReservations(){
        …
    }
}
```

- Handler methods do no longer need to be annotated with `@ResponseBody`

Spring 4 introduced the @RestController annotation, which is a convenience annotation combining the @Controller and the @ResponseBody annotations. So instead of annotating the class using @Controller and use the @ResponseBody annotation to bypass the ViewResolver, the class can now be annotation using just the @RestController annotation.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@RequestMapping

- Maps web requests to methods of Spring controller
- Can be used at class and method level
  - Class-level annotation maps request path to controller

```
@RestController
@RequestMapping("/rental")
public class ReservationController {   …   }
```

  - Method-level annotations make mapping more specific
    - Adding path information

```
@RequestMapping("/reservations")
public List<Reservation> getAllReservations(){
    return Collections.EMPTY_LIST;
}
```

    - Defining specific HTTP request types

```
@RequestMapping(path="/reservations", method=RequestMethod.GET)
```

- The `method` attribute does not define default type
  - When left undefined it maps to any HTTP request

To map URIs to the methods defined within the controller class, the @RequestMapping annotation is used. When used at class level, all method level mapping inherit from the class mapping, narrowing the definition for that specific method.

The @RequestMapping annotation does not define a default HTTP method (e.g. GET, POST). Instead when the method is not explicitly defined, the method will be used for all HTTP method types.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Request Mapping on Header (Values)

- Mapping can be narrowed for specific header
  - Request is mapped when header is (not) present

```
@RequestMapping(headers= {"applicationID"})
```

```
@RequestMapping(headers= {"!applicationID"})
```

  - Request is mapped when header value (not) equals
    - Header must be present

```
@RequestMapping(headers= {"applicationID=123"})
```

```
@RequestMapping(headers= {"applicationID!=123"})
```

- When used to check media type, wildcards are allowed

```
@RequestMapping(headers = "content-type=text/*")
```

To further narrow the URI mapping, the annotation allows the definition of headers to be (not) present. It is even possible to define a mapping which makes sure the method is only used when the header is present and (not) matches a certain value. When the content-type header is defined it is even possible to use the wildcard to define the possible values.

When using curl (a tool allowing to make different types of HTTP request from the command line) , the header value can be defined using the –H variable. Keep in mind that curl requires the header name and its value need to be separated using a colon -> curl -H "applicationID:123" http://localhost:8080/reservation/getAll

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Mapping Annotations

- Spring 4.3 defined new HTTP mapping annotations
  - Handling different HTTP request methods
    - @GetMapping
    - @PostMapping
    - @PutMapping
    - @DeleteMapping
    - @PatchMapping

```
@RestController
@RequestMapping("/rental")
public class ReservationController {
   @GetMapping("/reservation/{reservationID}")
   public Reservation getReservation(@PathVariable int reservationID) {
      …
   }
}
```

- Introduced to improve readability of code

Instead of using the @RequestMapping annotation and its 'method' attribute to define the HTTP method for which a method should be invoked, Spring 4.3 introduced several new HTTP mapping annotations.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@PathVariable and Template Patterns

- Template patterns can be used to define URI 'variables'
  - Template variable must be enclosed with curly braces

```
@GetMapping("/reservation/{reservationID}")
```

    - Multiple patterns can be defined

```
@GetMapping("/{location}/reservation/{reservationID}")
```

- @PathVariable binds value to method parameter(s)

```
@GetMapping("/reservation/{reservationID}")
public Reservation getReservation(@PathVariable("reservationID") String id) { … }
```

  - When parameter name matches name of path variable annotation value can be omitted

```
@GetMapping("/{location}/reservation/{reservationID}")
public Reservation getReservation(@PathVariable String reservationID,
                                  @PathVariable String location) {   …   }
```

When defining URIs template patterns can be defined within. When is request is made to the resource, the value used in the place of the variable can be bound to a method parameter of the controller method using the @PathVariable annotation.

When the name of the method parameter matches the name of the template pattern in the URI just adding the annotation is sufficient. When this is not the case the name of the template pattern must be defined within the annotation

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@PathVariable Types

- @PathVariable annotated parameters can be any simple type (e.g. String, int, long, …)
  - Spring automatically converts to the appropriate type
    - Throws TypeMismatchException when type not supported

```
@GetMapping("/reservation/{reservationID}")
public Reservation getReservation(@PathVariable int reservationID) { … }
```

- Annotated parameter of type Map<String, String> will be populated with all variable names and values

```
@GetMapping("/{location}/reservation/{reservationID}")
public Reservation getReservation(
   @PathVariable Map<String,String> uriVariables) {   …   }
```

- Custom Converter implementations can be defined

When the @PathVariable annotation is used to bind the value of the template pattern to the controller method, the type of the parameter must a simple type in order for the framework to properly map the incoming String into the parameter type. When the conversion cannot take place, a TypeMismatchException will be thrown.

When the type or the parameter is Map<String,String> it will automatically be populated with all the template patterns defined within the URI

When the parameter type of the controller mapping is not supported by default a custom implementation of the Converter interface can be created

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@PathVariable with Regular Expression

- Regular expressions can be used in URI template
  - Using {variable_name:regular_expression} syntax
- To match, URI variable must match regular expression
  - E.g. variable must be a numeric value
    - Avoiding NumberFormatException when mapping parameter

```
// URI /reservation/1000 matches
// URI /reservation/abc not matches
@GetMapping(path = "/reservation/{reservationID:\\d+}")
public Reservation getReservation(@PathVariable int reservationID) {   …   }
```

Just defining a template parameter in the URI would result in the method to be used for every value of this parameter. To restrict the mapping a regular expression can be added to the pattern.

For the controller method to be used, the value must exactly match the URI variable

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

More @RequestMapping

- Multiple paths can be mapped to same controller method
  - Value and path attributes accept multiple mappings

```
@GetMapping({"/reservation/{reservationID}", "/appointment/{reservationID}"})
public Reservation getReservation(@PathVariable int reservationID) { … }
```

- Method might handle multiple request types

```
@RequestMapping(method={RequestMethod.POST,RequestMethod.PUT})
```

Even though it might not be recommended, it is possible to define multiple URIs to a single method. The value and path attributes allow for multiple URIs to be defined.

The method attribute of the annotation allows the definition of multiple RequestMethod values when the method is to be used for multiple request types.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Wildcards in Path Mapping

- Wildcards can be used in path mapping
  - ? matches exactly one character in path segment

```
@GetMapping("/location?")
public ResponseEntity<?> getLoctationInformation() {   …   }
```

  - * matches 0 or more characters within path segment
    - Allows for definition of fallback method(s)

```
@GetMapping("*")
public ResponseEntity<?> fallback() {
   return ResponseEntity.noContent().build();
}
```

  - ** matches 0 or more path segments

```
@RequestMapping("**")
```

- Handling requests when no exact mapping was found

Wildcards can be used when defining URIs. A question mark in the URI matches exactly one character within a path segment, while an asterisk (*) matches zero or more characters within a path segment.

Two asterisk (**) can be used to define zero or more path segments within the URI

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The @RequestParam Annotation

- Request URLs might contain request parameters

```
http://localhost:8080/reservation/location?code=BOS
```

- Parameter values can be bound to method parameter

```
@GetMapping("/location")
public ResponseEntity<?> getLoctationInformation(@RequestParam String code)
```

  - Name of request parameter can be made explicit

```
@RequestParam("locationCode") String code
```

- By default referenced parameters are required
  - Can be made optional (results in null value)

```
@RequestParam(name = "locationCode", required = false)
```

  - A default value can be defined

```
@RequestParam(name = "locationCode", required = false, defaultValue = "NYC")
```

When making a request, request parameters can also be defined within the URL. To obtain the values of these parameters, the @RequestParam annotation should be added to the parameter of the controller method. When the name of the method parameter does not match the request parameter, the name of the request parameter can be explicitly defined within the annotation,

By default each @RequestParam annotated parameter is considered to be a required parameter and must therefore be present within the request URL. When the parameter might not be present, the parameter can be defined as optional and a default value can even be defined.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 12: Working with Spring REST

~/StudentWork/code/spring-rest-intro/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Introduction to REST Clients in Spring

REST principles
Introduction to RESTful Services in Spring
Introduction to REST Clients in Spring
Bootstrapping the REST application
Content Representation
Implementing the REST Service

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Introduce `RestTemplate` class
- Making GET, POST, PUT, HEAD, OPTIONS and DELETE requests
- Introduce the `UriTemplate` class
- Using `HttpEntity and RequestEntity`
- Use the exchange method to define 'complex' requests
- Process requests and responses using callback
- Configure the `RestTemplate`

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Java Applications as REST Clients

- Writing a REST client in Java can be arduous
  - Client application is responsible for
    - Creating an `HTTP` client
    - Creating a `URL` and an `HTTP` request
    - Sending the request and waiting for the response
    - Process the `HTTP` response
    - Handle `IOExceptions` that might be thrown
- HTTPClient API in Java 11 provides some relief
  - But much of the processing is relatively common for all REST clients

Writing a REST Client using the classes provided by the JDK has never been easy. This low-level API required developers to the deal with every aspect of the request and response.

A new HTTPClient API was introduced in Java 9 (and made official in Java 11) which provides simplified APIs, making it easier, but it would still require to program much of the boilerplate code that would be common to pretty much every REST client written in Java.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring's Support for REST Clients

- `RestTemplate` simplifies interactions with RESTful services
  - Defines methods for most common HTTP methods

```
RestTemplate restTemplate = new RestTemplate();
String result = restTemplate.getForObject(url, String.class);
```

- When message converters are available, they will automatically be registered
  - Allowing requests and responses to be converted to and from Java objects

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Reservation {
   private Integer reservationID;
   private String nameOnReservation;
   …
}
```

```
Reservation result = restTemplate.getForObject(url, Reservation.class);
```

  - `JsonIgnoreProperties` is part of Jackson JSON library

Spring provides full support for the development of REST client in Java. Most of the API evolves around the use of the RestTemplate class. This class defines a number of methods to send different types of requests to the service and contains references to the appropriate message converters to support the mapping between the Java type and the type used on the wire.

The template will automatically register converters that are made available to the application.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Making GET Requests

- Two methods can be used for getting resources
  - `getForObject()` and `getForEntity()`

```
<T> T getForObject(URI url, Class<T> responseType)
<T> ResponseEntity<T> getForEntity(URI url, Class<T> responseType)
```

- `getForEntity()` method returns `ResponseEntity`
  - Includes retrieved resource and additional information about the response
    - HTTP header information
    - HTTP status code
  - Retrieved resource is accessed using `getBody()` method

```
ResponseEntity<Reservation> response =
            restTemplate.getForEntity(url, Reservation.class);
HttpStatus statusCode = response.getStatusCode();
HttpHeaders headers = response.getHeaders();
Reservation body = response.getBody();
```

To make a GET request to the service the template provides two methods getForObject and getForEntity. While the getForObject returns 'only' the body of the response mapped to a Java object, the getForEntity returns a ResponseEntity, which, besides the message body, also provides access to the response status code and header information.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

URI Variables

- URI of resource might define URI variables

```
String url = "http://localhost:8080/rental/reservation/{reservationID}";
```

- Overloaded methods provide support for these variables

```
<T> T getForObject(String url, Class<T> responseType, Object... uriVariables)
<T> T getForObject(String url, Class<T> responseType,
                               Map<String, ?> uriVariables)
```

  - Using varargs to define values in correct order

```
Reservation response = restTemplate.getForObject(url, Reservation.class,2);
```

  - Using Map to define values for each named variable

```
Map<String, Object> uriVariables = new HashMap<>();
uriVariables.put("reservationID", 2);
Reservation response = restTemplate.getForObject(url, Reservation.class,
                                                 uriVariables);
```

The URIs that have been mapped to resource endpoint may contain URI variables. To define the values of these variables getXXX methods contain overloaded implementations that allow the definition of these values either as varargs or a Map.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Making a HEAD Request

- HTTP HEAD request is used to obtain only response headers
  - Similar to GET, but response body will not be retrieved
- Allows for fast checking of server information
  - E.g. checking last-modified date
- `RestTemplate` provides `headForHeaders` method
  - Returns `HttpHeaders` instance

```
HttpHeaders headers = restTemplate.headForHeaders(url);
MediaType contentType = headers.getContentType();
long contentLength = headers.getContentLength();
long lastModified = headers.getLastModified();
```

A HTTP HEAD request is made to the server to only retrieve the response header and not the body of the response. It is similar to making a GET request to the resource, except for the fact that the response body will not be retrieved. This approach could be used to check the last-modified date of a resource to determine whether or not the content should be refreshed.

To make a HEAD request to the server, the template class provides a headForHeaders method, which returns an instance of HttpHeaders.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Making POST Requests

- Creating new resources can be done using one of three methods
  - `postForLocation` method returns URI of new resource
    - Returns value of the `Location` header

```
ReservationData reservationData = …
URI location = restTemplate.postForLocation(url, reservationData);
```

  - `postForObject` return the new resource

```
Reservation reservation =
   restTemplate.postForObject(url, reservationData, Reservation.class);
```

  - `postForEntity` return `ResponseEntity` containing new resource

```
ResponseEntity<Reservation> entity =
    restTemplate.postForEntity(url, reservationData, Reservation.class);
```

To create a new resource, the HTTP POST method should be used. In addition to the postForObject and postForEntity method, a postForLocation method is also available which returns the URI of the resource that was created. (It returns the value located in the Location header of the HTTP response.)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Making an OPTIONS Request

- Allows client to determine which methods are available on resource
  - Response of request should be HTTP_OK (200) error code
- `RestTemplate` provides `optionsForAllow` method
  - Returns the `Allow` header value for a given URI

```
Set<HttpMethod> optionsForAllow = restTemplate.optionsForAllow(url);
```

Clients might query the service to determine what are the valid methods that can be used to communicate with the service. This can be accomplished by making an OPTIONS request, using the optionsForAllow method of the template.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Making a PUT or DELETE Request

- Updating resources is done using a PUT request

```
Reservation reservation = …
reservation.setNameOnReservation("Wilma Flinstone");
restTemplate.put(url, reservation);
```

- Removing resources is done using a DELETE request
  - Overloaded method support URI templates

```
String url = "http://localhost:8080/rental/reservation/{id}";
restTemplate.delete(url, reservationID);
```

  - Method does not return a value

The template also contains methods to support HTTP PUT and HTTP DELETE requests.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The `UriTemplate` Class

- Template is representation of URI containing variables

```
String template = "http://localhost:8080/rental/reservation/{id}";
UriTemplate uriTemplate = new UriTemplate(template);
```

- The `expand` method creates full API
  - Setting variable values using `varargs`
    - Defined in order of variables in URI

```
int reservationID = …
URI uri = uriTemplate.expand(reservationID);
```

  - Setting named variable values using Map
    - Key in Map represent names of variables in URI

```
Map<String, Object> uriValues = new HashMap<>();
uriValues.put("id", reservationID);
URI uri = uriTemplate.expand(uriValues);
```

Replacing the URI variables by their values can also be accomplished using the UriTemplate class. Once constructed using the URI, the full URI can be created by invoking the expand method, defining the values either as varargs or a Map containing key-value pairs.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The RestTemplate's exchange Method

- The exchange method can be used when specific details of request need to be defined
  - Defining HTTP header information
  - Defining generic types for return types
  - ...

```
restTemplate.exchange(url, method, requestEntity, responseType, uriVariables)
```

- Used for requests that are considered less common

```
String template = "http://localhost:8080/rental/reservation/{id}";
UriTemplate uriTemplate = new UriTemplate(template);
URI uri = uriTemplate.expand(reservationID);
HttpEntity<Void> httpEntity = …

ResponseEntity<Reservation> response =
      restTemplate.exchange(uri, HttpMethod.GET,
                            httpEntity, Reservation.class);
Reservation reservation = response.getBody();
```

When more details about the request need to be specified or when the return type of the invocation is a generic type, the exchange method of the template should me used.

Another option would be:

HttpEntity

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The HttpEntity and HTTPHeader Classes

- HttpEntity represents HTTP request or response
  - Consisting of both the headers and the body

```
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);
Reservation body = …
HttpEntity<ReservationData> entity = new HttpEntity<>(body, headers);
```

- HTTPHeader represents all HTTP header data
  - Mapping all header names to list of String values
    - Implementation of MultiValueMap<String, String>
  - Adds several convenience methods

```
headers.setContentType(MediaType.APPLICATION_JSON);
```

  - Defines constants for commonly used header names

```
String contentTypeHeaderName = HttpHeaders.CONTENT_TYPE;
```

An instance of HttpEntity represents both the headers and the body of a request or a response. The HTTPHeader class represents all header data that should be part of a request or response. Besides representing the MultiValueMap to hold all the header information it also contains several convenience methods for setting the most common header values. Additionally it contains a number of constants for commonly used values.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The `RequestEntity` Class

- Subtype of `HttpEntity`, adding method and URI info
  - Contains static utility methods for creating instances

Static methods of RequestEntity	
`HeadersBuilder get(URI uri)`	`BodyBuilder method(HttpMethod,URI)`
`HeadersBuilder delete(URI uri)`	`BodyBuilder patch(URI uri)`
`HeadersBuilder head(URI uri)`	`BodyBuilder post(URI uri)`
`HeadersBuilder options(URI uri)`	`BodyBuilder put(URI uri)`

- Request is built using convenience methods of builder

```
RequestEntity<Void> entity =
   RequestEntity.get(uri).accept(APPLICATION_JSON)
                         .ifModifiedSince(lastModified)
                         .build();
```

The RequestEntity class is a subtype of HttpEntity adding methods for defining the HTTP method and the URI of the request. Several convenience methods have been defined to simplify the creation of the request instance

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The `RequestEntity` Class (cont'd)

- Used by `exchange` method of `RestTemplate`
  - URL and HTTP method no longer need to be specified
    - Information is contained within `RequestEntity`

```
RequestEntity<Void> entity =
   RequestEntity.get(uri).accept(APPLICATION_JSON)
                         .ifModifiedSince(lastModified).build();
ResponseEntity<Reservation> response =
       restTemplate.exchange(requestEntity, Reservation.class);
```

- Can also be used as parameter in `@Controller` method
  - Providing access to all request data

```
@PostMapping
public ResponseEntity<Reservation> makeReservation(
                  RequestEntity<ReservationRequest> request) {

   HttpHeaders headers = request.getHeaders();
   ReservationRequest body = request.getBody();
   …
}
```

An instance of RequestEntity can now be used by one of the overloaded exchange methods of the RestTemplate. Since the RequestEntity already contains the URI of the request this information does no longer need to be provided to the exchange method.

RequestEntity can also be the type used by controller methods on the server side, providing the implementation to all information sent along with the request.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

ParameterizedTypeReference

- Java's generic types are not available at runtime
  - Compiler erases type information
- Obtaining type-safe (generic) collection instance is challenging

```
List<?> result = restTemplate.getForObject(url, List.class);
//returns List of LinkedHashMap instances
```

- `ParameterizedTypeReference` enables use of generic types at runtime
  - Type needs to be subclassed

```
ParameterizedTypeReference<List<Reservation>> type =
    new ParameterizedTypeReference<List<Reservation>>() { };
```

- Supported by overloaded `exchange` method of template
  - Not by convenience methods (e.g. `getForObject`)

```
ResponseEntity<List<Reservation>> result =
      restTemplate.exchange(url, GET, null, type);
```


Even though the introduction of generics in Java 5 simplified a lot of the APIs we know today, this generic information is not available at runtime. So when we expect a generic type as the result of the REST invocation, a subclass of ParameterizedTypeReference must be defined and provided to the exchange method.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The RestTemplate's execute Method

- The execute method can be used when specific details of a request need to be defined
  - Allows for manipulation of request and response using callback implementations

```
RequestCallback requestCallback = …
ResponseExtractor<ResponseEntity<Reservation>> responseExtractor = …

restTemplate.execute(url, POST, requestCallback, responseExtractor);
```

When details about the request need to be defined or detailed about the response need to be retrieved the execute methods of the template can be used. Managing both the request and the response is done by providing callback implementations.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The `RequestCallback` Interface

- Callback interface for Client-side HTTP requests
  - Allows for manipulation of both header and body

```
@FunctionalInterface public interface RequestCallback {
   void doWithRequest(ClientHttpRequest request) throws IOException;
}
```

- Implementation can be defined (e.g. using lambda)

```
RequestCallback requestCallback =  request -> {
      OutputStream os = request.getBody();
      //Convert bean to JSON and write to body of request
      ObjectMapper mapper = new ObjectMapper();
      mapper.writeValue(os, requestData);
      //Add content-type header
      request.getHeaders().add(HttpHeaders.CONTENT_TYPE, APPLICATION_JSON_VALUE);
   };
```

  - Or created using factory methods of `RestTemplate`
    - `acceptHeaderRequestCallback(Class)`
    - `httpEntityCallback(Object)`
    - `httpEntityCallback(Object, Type)`

The RequestCallback needs to be implemented at client side to define the callback handler for manipulating the request. Since the interface has been defined as a Functional Interface the callback can also be described using a Lambda expression.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The `ResponseExtractor` Interface

- Callback interface for retrieving data from response
  - Implementation does not have to manage resources

```
@FunctionalInterface
public interface ResponseExtractor<T> {
  T extractData(ClientHttpResponse response) throws IOException;
}
```

- Implementation can be defined (e.g. using lambda)

```
ResponseExtractor<ResponseEntity<Reservation>> extractor = response ->{
   InputStream is = response.getBody();
   HttpStatus statusCode = response.getStatusCode();
   HttpHeaders headers = response.getHeaders();
   …
};
```

- Or created using factory method of `RestTemplate`
  - `responseEntityExtractor(Type)`

An implementation of the functional ResponseExtractor interface is used to define the callback method for handling the response.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Define Connection Timeouts

- `RestTemplate` relies on `ClientHttpRequestFactory` for most configuration
  - By default uses `SimpleClientHttpRequestFactory` which defines infinite timeout settings
- Timeout settings can be customized programmatically

```
SimpleClientHttpRequestFactory clientHttpRequestFactory =
        new SimpleClientHttpRequestFactory();
clientHttpRequestFactory.setConnectTimeout(5_000);
clientHttpRequestFactory.setReadTimeout(5_000);

RestTemplate template = new RestTemplate(clientHttpRequestFactory);
```

- Default factory provides only limited functionality
  - Most often not sufficient for real-time applications

The default connection timeout used by the RestTemplate is set to infinite. To customize this setting a customized instance of the SimpleClientHttpRequestFactory can be supplied to the template.

Even though this implementation provides us with some default configuration options, this implementation is probably not sufficient to support real-time applications.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Advanced Template Configuration

- `HttpComponentsClientHttpRequestFactory` supports `HTTPClient` library
  - Supplies several useful features
    - Timeout settings
    - Connection pool configuration
    - Idle connection management
    - ...

```
private ClientHttpRequestFactory getClientHttpRequestFactory() {
   RequestConfig config = RequestConfig.custom()
                                       .setConnectTimeout(timeout)
                                       .setConnectionRequestTimeout(timeout)
                                       .setSocketTimeout(5000).build();
     CloseableHttpClient client = HttpClientBuilder.create()
                                                   .setDefaultRequestConfig(config)
                                                   .build();
     return new HttpComponentsClientHttpRequestFactory(client);
}
```

```
<dependency>
   <groupId>org.apache.httpcomponents</groupId>
   <artifactId>httpclient</artifactId>
</dependency>
```

HttpComponentsClientHttpRequestFactory is a more advanced implementation provided by the org,apache.httpcomponents.httpclient library and allows for a more low level configuration of the request factory.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 13: Implementing the Spring REST Client

~/StudentWork/code/spring-rest-client/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Bootstrapping the REST application

REST principles
Introduction to RESTful Services in Spring
Introduction to REST Clients in Spring
Bootstrapping the REST application
Content Representation
Implementing the REST Service

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Describe steps needed to bootstrap Spring REST application
- Configure Content Representation libraries
- Configure Spring MVC and map the Dispatcher Servlet
- Explain the advantages of using Spring Boot to setup the REST project
- Setup a Spring REST application using Spring Boot

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Bootstrap the REST Application

- Several steps need to be taken to setup a Spring REST application
  - Resolve library dependencies
    - Spring-MVC, Servlet API, …

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
</dependency>
```

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
</dependency>
```

  - Configure the Spring web application
    - Configure the `DispatcherServlet`
    - Map the servlet URL
    - …
  - Configuration of message converters

Several steps need to be taken when setting up a new REST application. Not only do we need to add the Spring MVC and servlet libraries to the classpath, the DispatcherServlet needs to be set up and the servlet URL needs to be defined. In addition, message converters responsible for mapping the Java types to and from JSON or XML need to be configured.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Handling Content Representation

- Resources should be represented in useful format
  - Requestor may provide JSON, XML, or some other format
  - Consumer may expect JSON, XML
    - ... or human readable form (HTML, PDF)
- Controllers work with Java objects
  - Not concerned with representation of resource(s) on wire
- Message converters need to be 'configured'
  - Mapping HTTP request and response to/from POJO
- jackson-databind library provides conversion between JSON and POJO

```
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
</dependency>
```

One of the core concepts of REST is the ability to represent resources in a variety of useful formats. When using Spring REST the actual mapping between the Java types that are being used by the controller methods and the data type that is sent over the wire is done by message converters. One of the libraries that could be used to map the Java type to JSON is the Jackson-databind library, which naturally must also be added to the build path of the application.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Configuring Spring-MVC

- `DispatcherServlet` is the entry point of all Spring-MVC applications
  - Intercepts and dispatches HTTP requests
- Servlet needs to be configured in the web application
  - Historically done in the web.xml
- Starting version 3.0 of Servlet API, web.xml became optional
  - Configuration of Spring-MVC can now be done using `WebApplicationInitializer` instance
    - All configuration can now be done using Java

```
public class ApplicationInitializer implements WebApplicationInitializer {
   @Override
   public void onStartup(ServletContext servletContext)
                                 throws ServletException { … }
}
```

The DispatcherServlet is the core of all Spring-MVC applications. It will intercept all requests that are made to the application and dispatches them to one of the various resources within the application. In order to accomplish this the servlet must be configured within the application and URIs need to be mapped to the Servlet.

Historically configuring servlets and their servlet mapping was done in the web.xml configuration file. With the introduction of the Servlet 3.0 specification this XML configuration file become optional and programmatic configuration become possible.

To configure a web application in Spring an implementation of the WebApplicationInitializer can be defined. Within the onStartup method of this interface the entire web application can be configured programmatically.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

AbstractAnnotationConfigDispatcherServletInitializer

- Implementation of `WebApplicationInitializer`
  - Simplifies configuration of dispatcher servlet
  - Registers `DispatcherServlet` instance

```
public class ApplicationInitializer
        extends AbstractAnnotationConfigDispatcherServletInitializer {
  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class[] { JavaConfig.class };
  }
  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class[] { WebApplicationConfig.class };
  }
  @Override
  protected String[] getServletMappings() {
    return new String[] { "/" };
  }
}
```

- Recommended approach for configuring Spring MVC

To simplify the configuration of the DispatcherServlet Spring provides the AbstractAnnotationConfigDispatcherServletInitializer class, which is an implementation of the WebApplicationInitializer interface.

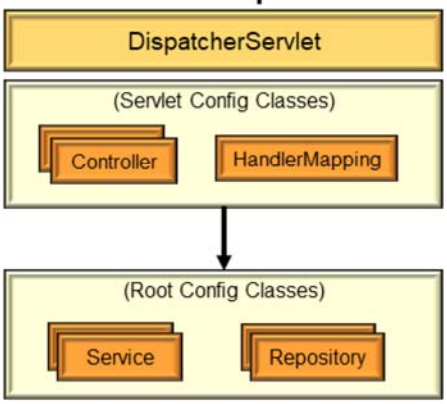
This utility class registers an instance of the DispatcherServlet with the web application. When extending this class, you must provide an implementation of the `getRootConfigClasses`, `getServletConfigClasses` and `getServletMappings` method to define the location of the Spring configuration classes and define the URLs to which the DispatcherServlet must be mapped.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Servlet and Root Config Classes

- Multiple Dispatcher servlets could be registered
  - Each Dispatcher has its own `WebApplicationContext`
- `ApplicationContexts` support hierarchical contexts
  - Root context defines common beans
    - Inherited beans can be overridden in sub-context
    - Beans can be defines for specific servlet instance



Keep in mind that in each Spring MVC application, multiple Dispatcher servlets could be configured, each servlet would have access to its own (unique) WebApplicationContext. As a result web resources could be defined within contexts specific to a single servlet.

However, several components (e.g. services) need to be available to all servlets within the application and their configuration should not have to be duplicated for each servlet.

The ApplicationContext supports the concept of hierarchical contexts. When a resource is needed, it will be looked up within the context. When it cannot be found in the current context the parent context will be queried for the component. By taking this approach, component definition cannot only be re-used by multiple contexts. Component definitions can also be overridden in a sub-context.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining the Servlet Application Context

- Configuration class defining web-related beans

```
@Configuration @EnableWebMvc
@ComponentScan(basePackages = "com.triveratech")
public class WebApplicationConfig {
   …
}
```

- `@EnableWebMvc` imports additional configuration class
  - Imports the (default) Spring MVC configuration
  - Detects Jackson and JAXB libraries on classpath
    - Automatically registers JSON and XML converters

```
@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.TYPE)
@Documented @Import(DelegatingWebMvcConfiguration.class)
public @interface EnableWebMvc {
}
```

- Only one `@Configuration` class may contain `@EnableWebMvc` annotation

To define a configuration for a Servlet application context the @Configuration class should be annotated using @EnableWebMvc. By adding this annotation to the configuration class additional configuration classes will be loaded into context, resulting the default configuration of Spring MVC and the automatic detection of Jackson and JAXB libraries on classes to register the message converters.

To make sure the configuration of Spring MVC is consistent within a context, only a single Configuration class should be annotated using @EnableWebMvc

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Customize Imported Configuration

- `@EnableWebMvc` annotated configuration classes may implement `WebMvcConfigurer` interface
  - Allows for customization of default configuration

```
@Configuration @ComponentScan(basePackages = "com.triveratech.endpoint")
@EnableWebMvc
public class WebApplicationConfig implements WebMvcConfigurer{
   @Override
   public void configureContentNegotiation(final ContentNegotiationConfigurer c) {
      Map<String, MediaType> mediaTypes = new HashMap<>();
      mediaTypes.put("xml", MediaType.APPLICATION_XML);
      mediaTypes.put("json", MediaType.APPLICATION_JSON);
      c.mediaTypes(mediaTypes);
   }
}
```

- `WebMvcConfigurerAdapter` provides stub implementation of interface methods
  - Deprecated in Spring 5
    - Interface methods now defined as default methods

To customize the default configuration of Spring MVC when adding the @EnableWebMvc annotation, the configuration class may implement the WebMvcConfigurer interface.

Until the introduction of Spring 5 the WebMvcConfigurerAdapter was often extend instead of implementing the interface to avoid having to implement all methods defined by the interface. However with the introduction of default methods in Java 8, Spring 5 defined all methods of the WebMvcConfigurer interface as default method and made the adapter class deprecated.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Starting with Spring Boot

- Spring Boot eliminates the pain of startup dependencies
  - Automatically registering Spring beans to setup framework
- Tooling creates a working project
  - Using smart defaults for beans found on classpath
  - Can be completely tailored for 'final' application
    - All defaults can be overridden
- Spring Boot impacts dependencies and configuration
  - Has no impact on programming model
- Spring Boot lets us focus on application functionality
  - Embraces *"convention over configuration"* approach

Spring Boot provides us with tooling to setup a new project. When developing an application (Spring based component) from scratch, the tooling can be used to create a new project which not only has all the required dependencies in place, but also has been configured to work out of the box. When we want to develop a new Spring-based web application, the tooling will generate a Maven (or Gradle) project containing all the library dependencies, some template classes and, most importantly, has been completely configured. The generated application is ready to be deployed on a web application. In the case of a web application, the generated application even contains an embedded web server (Tomcat), allowing us to run the web application from the command-line, without the need of deploying the application onto a web server before testing it.

Spring Boot relies of smart defaults. It is not dictating a setup, it configures some default settings but it still allows developers to override all of these settings. Most importantly, Spring Boot impacts the way we define project dependencies and project configuration. It does not require our application classes to implement specific interfaces. Spring Boot has no impact on the programming model.

Spring Boot completely focuses on 'Convention over Configuration'. Everything that can be configured by default will be configured. Only when those defaults do not apply to the application do we need to provide an alternative setup.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Boot Overview

- Provide easier mechanisms for Spring development
  - Manages library dependencies
  - Minimizing configuration of Spring
  - Provide embedded servers for testing
  - Work with Maven and Gradle
- Spring Boot attempts auto-configuration of Spring beans
  - Based on libraries found on build path
  - Based on absence of bean(s)
    - When bean is already present, it will not configure another
- Auto-configuration is non-invasive
  - All configuration can be made explicit
    - Replacing automatically configured beans

One of the goals of Spring Boot is to allow a developer to quickly setup a new Spring project, without having to worry about selecting the correct libraries, setting up data-sources and defining templates. In other words, Spring Boot allows developers to once again focus on the implementation of the business logic and not waste time writing all this boilerplate code and configuration. In addition to providing a quick an easy way to setup a new project, Spring Boot also provides a number of 'tools' that allow us to monitor and measure applications at runtime.

Beans can be registered when a specific class is present on the classpath, a bean of a certain type hasn't already been registered in context or when a file exists on a given location.

Spring Boot attempts to register beans it finds on the classpath, using smart defaults. For example, when the Spring-JPA library is found on the classpath, Spring Boot attempts to register the HibernateTemplate (and all beans required to support this template).

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The Maven POM File

- Each 'technology' needed results in starter dependency
  - Downloads all necessary dependencies
  - Starter defines spring-webmvc module

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

- Library versions are determined by the parent pom
  - Downloaded from Spring repository
  - All dependency versions are tied to Spring Boot version
    - Guaranteed to be supported combination of library versions

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.1.2.RELEASE</version>
</parent>
```

The POM file contains a spring-boot-starter-… dependency for each of the required dependencies. The library versions that are added to the classpath are tied to the Spring Boot version that was selected.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

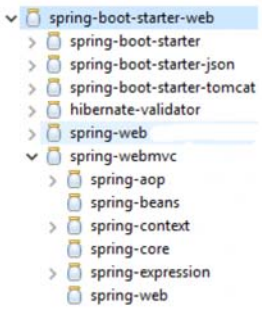
Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Starter spring-boot-starter-web

- Adding spring-boot-starter-web dependency
  - Automatically configures DispatcherServlet

```
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass(DispatcherServlet.class)
@AutoConfigureAfter(ServletWebServerFactoryAutoConfiguration.class)
public class DispatcherServletAutoConfiguration {
   …
}
```

  - Defines default error pages
  - Defines default content negotiation
  - Adds embedded Tomcat container
    - Allowing application to run on embedded server
  - …



Adding the start-web dependency to the POM file results in the correct libraries to be added to the build path. It also results in the auto-configuration of the DispatcherServlet.

All auto-configuration logic is implemented in spring-boot-autoconfigure.jar. WebMvcAutoConfiguration is defined in the /META-INF/spring.factories file of this jar.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The Bootstrap Class

- Bootstrap Application class serves two purposes
  - Primary configuration class
  - Bootstraps the application

```
@SpringBootApplication
public class Application {
   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }
}
```

- `@SpringBootApplication` enables component scanning and auto-configuration
  - Combination of three annotations
    - `@Configuration`
    - `@Componentscan`
    - `@EnableAutoConfig`
    - By default, it scans all components in same package or below

An application that was defined using Spring Boot may also contain an Application class, which serves two purposes. It acts as the primary configuration class of the Spring Boot application and is the bootstrap class, allowing the application to be started from the command-line.

The Application class has been annotated using the @SpringBootApplication, which serves the same purpose as adding @Configuration, @EnableAutoConfiguration and @ComponentScan

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Boot Maven Plugin

- Spring Boot Maven Plugin provides Spring Boot support in Maven
  - Allows for creation of executable jar archive
  - Allows application to run using embedded server

```
<plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

  - Requires Maven 3.2 (or newer)

In addition to the Maven dependencies to add libraries and auto-configuration, a Boot plugin can also be added to the build file, allowing the application to be packaged as an executable jar file and to run using an embedded server implementation.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Bootstrap Spring REST

- Spring Boot takes care of common configuration
  - Defining requires library (jar) files
  - Configuration of DispatcherServlet
  - ...
- All Spring Boot defaults can be overridden
  - Configuration can be fully customized
- Spring Boot provides embedded Tomcat container
  - Allowing executable jar to run on embedded server
- Developer can focus on implementation of REST endpoint
  - Instead of configuration of framework

So instead of doing all the configuration of Spring MVC manually Spring Boot takes care of most of the most common configuration.

Spring Boot defines default values for most of the configuration settings of a Spring MVC application, allowing for developers to focus in the business specific logic instead of having to worry about the configuration of the framework.

All default configuration however can be overridden within the application.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 14: Spring Hotel Reservation

~/StudentWork/code/spring-hotel-reservation/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 15: Spring Hotel Reservation Client

~/StudentWork/code/spring-hotel-reservation-client/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Content Representation

REST principles
Introduction to RESTful Services in Spring
Introduction to REST Clients in Spring
Bootstrapping the REST application
Content Representation
Implementing the REST Service

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Returning different media types from service
- Introduce negotiated resource representation
- Configure Message Converters

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Returning Multiple Representations

- REST implementation might expose multiple representations of resources
  - JSON, XML, HTML, PDF, ...
- Requested media type can be determined
  - Using Accept header in the request
  - Using URL suffixes in the request
  - Using URL parameter in the request
- HTTP `Content-Type` header indicates type of request body
- HTTP `Accept` header indicates expected type of response

A REST endpoint might provide resources in a variety of representations. While some of them are intended to be processed by computers (XML, JSON) the endpoint might also return the resource in a human readable format.

The requested media type can be determined using three distinct approaches. Using the Accept request header, using the suffix (extension) of the URI used to make the request or my using a request parameter.

Keep in mind that the Content-Type header of a request indicated the type of the content contained within the body of the request while the Accept header of the request defines the expected return type.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Controller Implementations

- Controller implementation is not aware of type used for communication
  - Mapping transport type to/from Java type is done by Spring

```
@GetMapping("/reservation/{reservationID}")
public Reservation getReservation(@PathVariable int reservationID) {
   …
   return service.getReservation(reservationID);
}
```

- Java class should follow JavaBean convention
  - Similar to classes used in JPA or JAXB
    - Java Persistence API (JPA) supports interoperability between object model and relational model
    - Java Architecture for XML Binding (JAXB) supports interoperability between XML's hierarchical model and object model

The controller implementation is not aware of any of the media types that is presented or requested by the client. The implementation relies on the regular Java object to do its job and message converters to perform the mapping between the media type and the Java type.

To support the conversion the Java classes that represent the request and response body should follow the JavaBean conventions

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Handling Transformations in Spring

- Spring provides two mechanisms for transforming a Java representation into a client-friendly representation (and vice versa)
  - Negotiated view-based rendering
    - Used to handle transformations into a client-friendly representation
      - Typically a human readable representation
  - HTTP message converters
    - Used to handle transformations from a client-supplied representation
      - Typically not a human readable representation
    - Used to handle transformations to a client-friendly representation

Spring provides the mechanisms for the transformation to and from the Java representation. It does so through the use of ViewResolver(s) to determine the best view for the request and message converters that perform the conversion of client-friendly representations to something the service can process and transforming the response into some client-friendly representation again.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Negotiated Resource Representation

- Spring REST uses `ContentNegotiatingViewResolver`
  - Determines best view for requested representation
- View resolver can be configured by implementing `WebMvcConfigurer` interface
  - Implementing `configureContentNegotiation()`

```
@EnableWebMvc @Configuration
public class RESTConfig implements WebMvcConfigurer {
   @Override
   public void configureContentNegotiation(ContentNegotiationConfigurer config) {
      config.ignoreAcceptHeader(true).favorPathExtension(true)
            .defaultContentType(MediaType.APPLICATION_JSON)
            .mediaType("xml", MediaType.APPLICATION_XML);
   }
}
```

- Configuration must be annotated with `@EnableWebMvc`

Resolving the representation type in Spring REST is done by an instance of the ContentNegotiatingViewResolver. When this process needs to be modified, the resolver can be configured by having a @EnableWebMvc annotated configuration class implement the WebMvcConfigurer interface and implement the configureContentNegotiation method.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Determining Representation Client Wants

- `ContentNegotiatingViewResolver` looks is for an extension at the end of the URI
  - Will be mapped to `application/<extension>` media type
    - Regardless of value in `Accept` header
- When extension is not found, `Accept` header is used
  - Browsers often hard wire value of Accept header
    - Not reflecting type the client really wants
- When no type is selected from `Accept` header `defaultContentType` is used

By default the ContentNegotiatingViewResolver will look for an extension at the end of the URL to determine the media type. So when the URL ends with the extension .json, the selected media type will be application/json. When the URL contains an extension, the value defined by the Accept header of the request will be ignored. When the URL does not contain an extension the value of the Accept header is used.

Since most browsers use a hard-coded value for this Accept header, the response does not always reflect the actual type the user is interested in.

When no media type can be determined , the defaultContentType value will be used.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Message Converters

- Available converters handle interoperability between Java and various representations
  - Converting message body depending on MIME type
    - Convert request body to a Java class
    - Convert controller response to MIME type
- Several implementations are provided by Spring
  - Automatically registered by `RestTemplate` and Spring MVC

```
public interface HttpMessageConverter<T> {
   boolean canRead(Class<?> clazz, MediaType mediaType);
   boolean canWrite(Class<?> clazz, MediaType mediaType);
   List<MediaType> getSupportedMediaTypes();
   T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
                              throws IOException, HttpMessageNotReadableException;
    void write(T t, MediaType contentType, HttpOutputMessage outputMessage)
                              throws IOException, HttpMessageNotWritableException;
}
```

The conversion between the data that is transmitted over the wire and the Java object that is used to represent the request or response is done by Message Converters. The converters that is being used depends on the mime type that is defined by the content-type header of the request or by the media type that is being requested by the client.

Several implementations of the message converters are being provided and registered by Spring when using the RestTemplate or the Spring MVC framework.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Standard Message Converters

- Spring comes with set of 'standard' converters
  - `StringHttpMessageConverter`
    - Reads all media types into `String` (`text/plain`)
  - `FormHttpMessageConverter`
    - Handles `application/x-www-form-urlencoded` (writes `multipart/form-data`)
  - `ByteArrayHttpMessageConverter`
    - Reads all media types (writes `application/octet-stream`)
  - `SourceHttpMessageConverter`
    - Converts javax.xml.transform.Source

Several converters are available by default, supporting the conversion between some basic media types (String, bytes, HTML forms, etc.)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

'Optional' Message Converters

- Several converters are only enabled when implementations are added to classpath
  - `Jaxb2RootElementHttpMessageConverter`
    - Converts Java objects to/from XML (requires JAXB2 on classpath)
  - `MappingJackson2HttpMessageConverter`
    - Converts Java objects to/from JSON (requires Jackson 2 on classpath)
  - ...
- Spring loops through registered message converters
  - Seeking first that fits given MIME type and class
  - Controller method must be annotated with `@ResponseBody`
    - Or class must be annotated using `@RestController`

Several other converters have been defined by Spring, but they are only applied when the appropriate converter implementation is added to the classpath of the application. So in order to support the conversion between Java and JSON, or between Java and XML, additional libraries need to be added to the build path.

When a request or response needs to be handled by Spring, it will iterate through the available message converters. The first converter which is capable of handling the requested mime type will be used.

Keep in mind that in order for Spring to use the Message Converter, the controller method must be annotated using the @ResponseBody annotation. (Or the controller class should be annotated using the @RestController annotation, which implicitly adds the @ResponseBody annotation to the methods.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Customizing Message Converters

- Converters can be customized by implementing `WebMvcConfigurer`
  - Overriding `configureMessageConverters` method

```
@EnableWebMvc
@Configuration
public class RESTConfig implements WebMvcConfigurer {
   @Override
   public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
      MarshallingHttpMessageConverter xmlConverter =
                          new MarshallingHttpMessageConverter();
      //configuring xmlConverter here
      …
      converters.add(xmlConverter);
      WebMvcConfigurer.super.configureMessageConverters(converters);
   }
   …
}
```

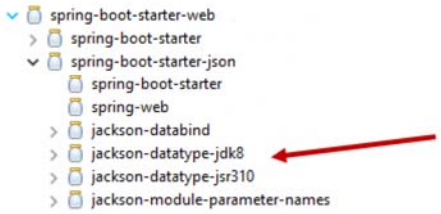
When the order of the converters needs to be modified, or the converter requires more low level configuration, the converter instance can be configured by having the controller class implement the WebMvcConfigurer interface and overriding the configureMessageConverters method.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

JSON Converter

- When using Spring Boot, Jackson for JSON is automatically added



  - Adding `MappingJackson2HttpMessageConverter`
- Otherwise library needs to be added to classpath

```
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
</dependency>
```

When bootstrapping the REST application using Spring Boot, the Jackson library is added to the classpath by default. As a result the, JSON converter is automatically added registered. When Spring Boot is not being used, the Jackson library has to be manually added to the classpath in order to support the conversion between JSON and Java.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

XML Converter

- When XML media is required a library needs to be added
  - Could be Jackson XML API

```
<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

  - Could be JAXB implementation

```
<dependency>
   <groupId>javax.xml.bind</groupId>
   <artifactId>jaxb-api</artifactId>
</dependency>
```

    - ...would require JAXB annotations to be added to beans

```
@XmlRootElement
public class Reservation implements Serializable {   …   }
```

To support the conversion between an XML payload and Java instances, an XML converter needs to be enabled. Spring allows the use of the Jackson library or the JAXB library to accomplish this.

When using the JAXB implementation, the Java beans that are being used by the controller methods do need to be annotated with the JAXB annotations.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The URL Suffix Strategy

- By default Spring checks path extension to determine requested media type
  - Extension is mapped to `application/<extension>` media type

```
http://localhost:8080/reservation/1001.xml
```

- When extension is not found, `Accept` header is used

The default approach of Spring for determining the requested media type is by looking for an extension in the URL of the request. The extension of the URL is then mapped to the application/<extension> media type. When no extension is found in the URL of the request, Spring will look for the value of the Accept header.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Using the `Accept` Header

- When making request, Accept header can be defined using `RestTemplate`

```
RequestEntity<Void> request =
          RequestEntity.get(uri).accept(MediaType.APPLICATION_JSON).build();
ResponseEntity<String> result = restTemplate.exchange(request, String.class);
```

- Requested type must be supported by server
  - Otherwise HTTP 406 (Not Acceptable) is returned
- Server can be configured to ignore Accept header

```
public void configureContentNegotiation(ContentNegotiationConfigurer config) {
   configurer.ignoreAcceptHeader(true);
}
```

When using the RestTemplate to programmatically make a REST request, the value of the Accept header can be defined my invoking the accept methods of the RequestEntity.

When the requested media type is not supported by the server an HTTP 406 error code will be returned by the server. When the server should completely ignore the value of the Accept header, the ContentNegotiationConfigurer can be configured in the @Configuration class.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Using Parameter to Define Media Type

- Requested media type can also be defined using request parameter
  - Support has to be enabled using `favorParameter` method
  - Accept header must be ignored!

```
@EnableWebMvc @Configuration
public class RESTConfig implements WebMvcConfigurer {
   @Override
   public void configureContentNegotiation(ContentNegotiationConfigurer config) {
      config.ignoreAcceptHeader(true).favorParameter(true);
   }
}
```

- Default parameter name is 'format'

```
http://localhost:8080/reservation/{id}?format=xml
```

  - Can be changed using `parameterName` method

```
config.ignoreAcceptHeader(true).favorParameter(true).parameterName("media");
```

When the Accept header is being ignored and the URL does not contain an extension, the media type can also be defined by adding a request parameter to the request. The server needs to be configured to not only ignore the accept header, but the favorParameter method should be invoked to enable this feature.

By default the request parameter that hold the media type is named ‘format’. The name of this parameter can be modified when configuring the ContentNegotiationConfigurer using the parameterName method

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining Parameter Media Types

- When using parameter strategy, media types should be mapped to key
  - Parameter value should match one of keys

```
public void configureContentNegotiation(ContentNegotiationConfigurer config) {
  config.ignoreAcceptHeader(true).favorParameter(true)
  .parameterName("media")
  .mediaType("PDFDocument", MediaType.APPLICATION_PDF)
  .mediaType("ExtensibleMarkupLanguage", MediaType.APPLICATION_XML);
}
```

```
http://localhost:8080/reservation/{id}?format=ExtensibleMarkupLanguage
```

When the parameter approach is being used to determine the media type, the value of this parameter needs to be mapped to a valid media type.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Handling the Request

- When method parameter is annotated with `@RequestBody`, message converters will be applied
  - Representation of request body should match specified value of `Content-Type` header
    - If `Content-Type` headers do not match, method will not be invoked

```
@PutMapping(path = "/{id}", consumes = "application/json")
public ResponseEntity<Reservation> updateArticle(
        @PathVariable int id,
        @RequestBody Reservation reservation) {

   Reservation updated = service.updateNameOnReservation(reservation);
   if (updated != null) {
      return ResponseEntity.ok(updated);
   }
   return ResponseEntity.notFound().build();
}
```

When a parameter of the controller method has been annotated using @RequestBody a message converter will be applied to convert the body of the request to the type of the parameter.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Mapping by Media Type

- Mapping can be defined by media type, produced or consumed by method(s)

```
@RestController
public class ReservationController {
@GetMapping(path = "/reservation/{id}",
            produces = { "application/json" })
public Reservation getReservationJSON(@PathVariable int id) { … }

@GetMapping(path = "/reservation/{id}",
            produces = { "application/xml" })
public Reservation getReservationXML(@PathVariable int id) { … }
}
```

- Controller now only handles requests for media type of `text/xml` or `application/json`
  - When other type is requested HTTP response code 406 (not acceptable) is returned

Mapping different media types to different methods within the controller can be done defining media types as values of the the produces (and consumes) attributes of the mapping annotations.

By doing so, the request is only mapped to the mapped when both the URL and the media type matches the type defined by the annotation.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 16: Spring REST Content Negotiation

~/StudentWork/code/spring-rest-content-client/lab-code
~/StudentWork/code/spring-rest-content/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Implementing the REST Service

REST principles
Introduction to RESTful Services in Spring
Introduction to REST Clients in Spring
Bootstrapping the REST application
Content Representation
Implementing the REST Service

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Process for Spring REST Implementation
- The Domain object
- Using Project Lombok to define the domain object
- (Not) Using Data Transfer Objects
- `ResponseEntity` builder interfaces
- Setting Location header using `UriComponentsBuilder`

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Process for Spring REST Implementation

- Parallels the design process
  - Develop or identify data representations
  - Develop the service
    - Define the service class
    - Bind the service class to a relative root path
    - Bind incoming HTTP requests to methods to service the requests
  - Configure the service
    - Identify the service lifecycle of singleton or per-request

The process for the implementation of the service parallels the design process of the application. We will start by identifying and developing the data representations. Once these representations have been developed the service class can be implemented and bound to a root path. Next the operations that need to be made public can be mapped to methods within the service class.

Once the service has been developed it should be determined whether the service instance can run as a singleton or a new instance is needed for each incoming request.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining Operations of the Resource

- Defining the 'usual' crud operations
  - Defining return values for both success and failure

HTTP Method	URL	HTTP Status	Description
GET	/reservations	200 (OK)	Returns list of all reservations in system
GET	/reservations/{id}	200 (OK) 404 (NOT Found)	Returns reservation for given reservation ID 404 error code is returned when id is not found
POST	/reservations	201 (created)	Add a new reservation. Body of request should contain reservation data
PUT	/reservations/{id}	200 (OK) 404 (Not Found)	Updates existing reservation. When reservation is not found or 404 is returned. On success, updated reservation is returned
DELETE	/reservations/{id}	204 (No Content) 404 (Not Found)	Delete reservation for the given id. 404 error code is returned when id is not found

When defining the operations that can take place on the resource, it is important to model the outcome of the operation for both success and failures. The table shown above provides an overview of all operations that are defined for the reservation resource, defining what each operation will return and what code will be returned to the client when the operation fails.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining the Resource

- Resource class can be Plain Old Java Object (POJO)
  - Following JavaBean conventions
    - Getter and Setter methods
    - Default (no-argument) constructor
    - …
- Instances op domain classes should be identifiable
  - Each (unique) instance should have a unique ID

```
public class Reservation{
   private Integer reservationID;   //Generated value
   private String reservationNumber;//Assigned by application
   private String nameOnReservation;
   private String rentalLocation;
   private LocalDate rentalDate;
   private String returnLocation;
   private LocalDate returnDate;
   private RentalCategory category;
   //getter and setter methods
}
```

The resource is represented by the Plain Old Java Object and should follow the JavaBean conventions. REST defines that each resource should be identifiable. Each instance of the resource should have a unique ID.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Introduction to Project Lombok

- Java is often considered to be too verbose
  - Lots of boilerplate code is required
- Classes often require implementation of
  - One or ore constructors
  - Getter and Setter methods for accessing instance variables
  - Implementation of `equals` and `hashCode` methods
  - Defining implementation of `toString` method
- Methods are added to comply to specifications and/or frameworks
  - Following the Java Bean convention
  - Allowing instances to be properly managed by collections
  - Allowing bean to be used by JPA or JAXB
  - ...

When looking at a Java class, specially when the class represents an entity, you will soon discover that there is a lot of boilerplate code within the class, in addition to the properties that actually define the entity.

Classes require the definition of one or more constructors, accessor methods to get and set the values of the instance fields, equals and hashCode in order to be able to properly manage entities within a Collection implementation and a toString method to allow for better debugging of code.

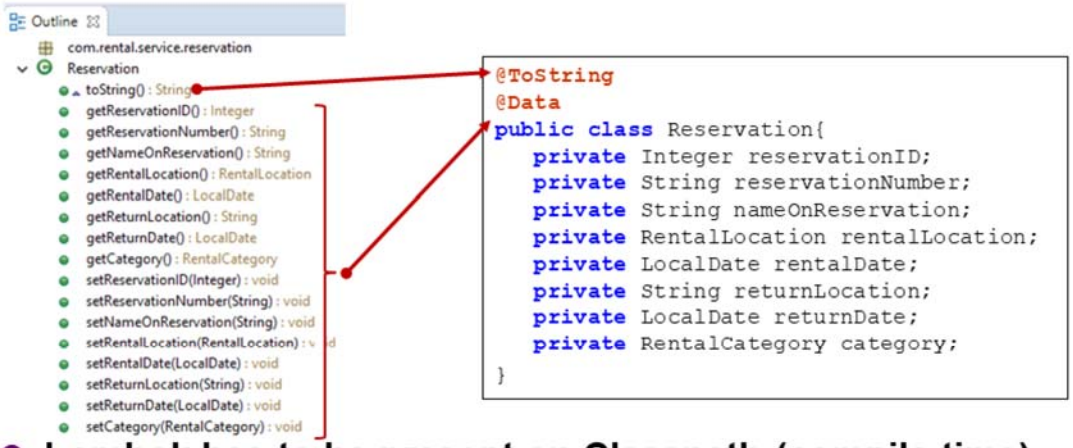
Several frameworks and specifications actually require classes to provide implementations of this methods in order to be able to use these instances within that framework.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lombok Annotations

- Lombok reduces boilerplate code for model/data objects
  - Using annotations and annotation processors



- Lombok has to be present on Classpath (compile time)

```
<dependency>
   <groupId>org.projectlombok</groupId>
   <artifactId>lombok</artifactId>
   <scope>provided</scope>
</dependency>
```

Lombok allows for annotations to be added to the classes, methods and fields. In addition an annotation processor provided by Lombok will use these annotations to generate methods and constructors which developers would normally program manually.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

(Not) Returning Entities in Controller

- Controller could return instances of entity class
  - Result would not include links to related resources
  - Would not allow for response headers to be defined
  - Couples application layers
    - Changes to service layer affects other application layers
  - Should consider lazy-loading aspects of Spring Data
    - Not all object associations are eagerly fetched

```
@RestController
public class ReservationController {
   @GetMapping("reservations")
   List<Reservation> getAllReservations() { … }
   @GetMapping("/reservations/{reservationID}")
   Reservation getReservation(@PathVariable Integer reservationID) { … }
}
```

The methods that are being defined within the controller class could just return instances of an entity class, but keep in mind that REST is more than just returning the resource. A proper REST resource returns the requested data and the links to related resources.

Another drawback of exposing entities in a remote interface is that is results in a very tight coupling between the different layers of the application. Not only are you exposing the internals of the application, when changes need to be made to the application, this change is not limited to the application itself, but also trickles down to the public interface of the application.

Last but not least, when using a ORM framework, not all object dependencies are eagerly fetched when the data is loaded from the database. The state of the 'root' object might be loaded, but object instances referenced might not be loaded until its property is accessed. To accomplish this a transaction to the database must still be available. Since transaction scopes are often defined on the service layer and not the presentation layer, by the time the entity is returned to the client, no database connection is available to load the object associations.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Data Transfer Objects (DTO)

- Carries data between processes
  - Used as parameter in (remote) interfaces
- Advantages of using DTO
  - May reduce number of remote calls
    - DTO can be populated using multiple entities
  - REST resources can expose subset of entity attributes
  - Persistent entities do not require non-persistent related annotation
- Disadvantages of using DTOs
  - Increased development time due to additional classes
  - Boilerplate code needed to facilitate mapping between entities and DTO

A Data Transfer Object is an object used to carry state between processes. In general a DTO only defines state and little or no behavior and is used as parameters and return values of remote interface. By using DTOs there is a clean separation between in internal entity data and the values that are returned to the client, allowing for not only a subset of data to be returned, but could also include conversions of data types. Using DTOs might also reduce the number of remote calls that need to be made to obtain all information, since a DTO can be populated from multiple entities which would normally have to be requested individually.

Another advantage of using DTOs that any annotation meta-data in both the entity and DTO is related to a single layer. In other words, an entity only contains annotations that deal with the mapping between the entity and the persistent store, it does not contain annotations needed to map the object to (for example) JSON

The disadvantage of using DTOs is that it requires developers to create more classes, code needed to perform the conversion between the DTO, and the entity needs to be added to the controller.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Using DTOs

- Project Lombok help to simplify development of DTO
  - Prevents from writing getter/setter, toString, etc.
- Mapping between DTO and Entity can be accomplished using mapping frameworks
  - ModelMapper
  - MapStruct
  - Dozer

When using DTOs, tooling like Project Lombok can help to limit the time needed to develop and maintain the extra classes. To perform the mapping between the DTO and the Entity several Java mapping frameworks exist, making sure that the conversion between these two object can (in most cases) be limited to a single line of code.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Using Mapper Implementations

- Mapping between source and destination type
  - Based on matching strategies and additional mapping configurations

```
@Entity @Table(name = "RESERVATIONS")
@Data
public class Reservation {
   @Id
   private Integer reservationID;
   private String nameOnReservation;
   …
   @ManyToOne
   private RentalLocation rentalLocation;
}
```

```
@Entity @Table(name = "LOCATIONS")
@Data
public class RentalLocation {
   private String locationID;
   private String state;
   …
}
```

```
@Data
public class ReservationDTO {
   private String reservationNumber;
   private String nameOnReservation;
   private String locationID;
   private String rentalLocationState;
   private LocalDate rentalDate;
   private String returnLocation;
   private LocalDate returnDate;
   private String category;
}
```

```
Reservation reservation =  …
ModelMapper modelMapper = new ModelMapper();
ReservationDTO dto = modelMapper.map(reservation, ReservationDTO.class);
```

In the example shown above two JPA entities are used by the implementation of the service. A uni-directional Many-to-One relationship is even defined between the Reservation and the RentalLocation instance.

A single ReservationDTO class defines the Data Transfer Object that is used while communicating with a remote client. Notice how the DTO defines properties that need to be taken from both the Reservation and RentalLocation enties.

Mapping frameworks rely on naming strategies (and optionally some custom configurations) to perform the mapping between two instances. In this case we are using the ModelMapper APIs map method that perform the actual mapping between the two

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

ResponseEntity

- Represents entire HTTP response
  - Status code, response headers and response Body
- Endpoint returns instance of `ResponseEntity`
  - Allows for fine-tuning of HTTP response
    - Defining status codes

```
@GetMapping("reservations")
ResponseEntity<List<Reservation>> getAllReservations() {
   List<Reservation> reservations = service.getAllReservations();
   List<ReservationDTO> dtos = reservations.stream()
         .map(r -> modelMapper.map(r, ReservationDTO.class))
   return new ResponseEntity<>(dtos, HttpStatus.OK);
}
```

    - Setting header values

```
List<ReservationDTO> dtos = …
HttpHeaders headers = new HttpHeaders();
headers.add("Cache-Control", "max-age=60");
return new ResponseEntity<>(dtos, headers, HttpStatus.OK);
```

The ResponseEntity class represents the entire HTTP response, so not only doe it hold the resource that needs to be returned, it also allows for the definition of the status code and response headers.

The methods now no longer return an instance of the Resource class, instead they return and instance of ResponseEntity, allowing developers to fine-tune the HTTP response before returning

Several overloaded constructors are available to create instance of the ResponseEntity

`ResponseEntity(HttpStatus status)`

`ResponseEntity(MultiValueMap<String,String> headers, HttpStatus status)`

`ResponseEntity(T body, HttpStatus status)`

`ResponseEntity(T body, MultiValueMap<String,String> headers, HttpStatus status)`

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

ResponseEntity Builder Interfaces

- Static methods provide access to builder interfaces
  - Simplify creation of ResponseEntity

```
@GetMapping("/reservations/{id}")
public ResponseEntity<ReservationDTO> getReservation(@PathVariable Integer id) {
   if (id < 1)
      return ResponseEntity.badRequest().build();
   …
}
```

ResponseEntity.BodyBuilder

Method	HTTP Status
accepted()	202
badRequest()	400
created(java.net.URI location)	201
ok()	200
status(HttpStatus status)	<<status provided>>
status(int status)	<<status provided>>
unprocessableEntity()	422

ResponseEntity.HeadersBuilder<?>

Method	HTTP Status
noContent()	204
notFound()	404

To simplify the creation of the ResponseEntity several static methods on the class provide access to the builder interfaces.

Methods returning HeadersBuilder

allow(HttpMethod... allowedMethods) -> Defines Allows header

cacheControl(CacheControl cacheControl) -> Set caching directive (Cache-Control header)

eTag(String etag) -> Defines ETag header

header(String headerName, String... values) -> Set given header

headers(HttpHeaders headers) -> Add headers to header map

lastModified(…) -> Set time when resource was last changed

location(java.net.URI location) -> Set the location of the resource (Location header)

Methods returning BodyBuilder

contentLength(long contentLength) -> The content of the body in bytes (Content-Length header)

contentType(MediaType contentType) -> Media type of the body (Content-Type header

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

ResponseEntity Builder Interfaces (cont'd)

- HeadersBuilder allows for definition of various response headers
  - Cache-Control, Last-modified, …
- The build() method creates response without body

```
return ResponseEntity.notFound().build();
```

- BodyBuilder is subtype of HeaderBuilder
  - Adds methods to define body, content-type and content-length
- The body(T body) methods sets body of response

```
@GetMapping("/rentalcar/image")
ResponseEntity<byte[]> getCarImage(RentalCategory category) {
   byte[] media = …
   return ResponseEntity.ok()
                        .contentLength(media.length)
                        .contentType(MediaType.IMAGE_PNG)
                        .body(media);
}
```

The noContent and notFound methods return an instance of HeaderBuilder, which allows us to modify only the headers of the response. Once all header information has been set, the build method needs to be invoked to create the ResponseEntity instance.

The other static methods return an instance of BodyBuilder. This subtype of HeaderBuilder adds methods that allow developers to controller the body of the response. Once the response has been properly configured the body method needs to be invoked, supplying the body of the response to create the ResponseEntity instance.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Creating the ResponseEntity

- Shortcut methods exist to create ResponseEntity
  - Returning body with status OK

```
@GetMapping("reservations")
public ResponseEntity<List<ReservationDTO>> getAllReservations() {
   List<ReservationDTO> dtos = …
   return ResponseEntity.ok(dtos);
}
```

  - Returning status OK (with body) or NOT_FOUND (with empty body) when Optional is empty

```
@GetMapping("/reservations/{id}")
public ResponseEntity<ReservationDTO> getReservation(@PathVariable Integer id) {
   Optional<Reservation> reservation = …
   Optional<ReservationDTO> dto =
           reservation.map(r -> modelMapper.map(r,ReservationDTO.class));
   return ResponseEntity.of(dto);
}
```

Static shortcut methods have also been defined to simplify the creation for the ResponseEntity instance when no details need to be added to the response. The static OK method can be used to return the response with static OK, while the ‘of’ method returns a status OK or NOT_FOUND. The of method accepts an instance of Optional (introduced in Java 8). When the optional is empty NOT_FOUND is returned, otherwise the content of the Optional is returned with status OK

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

REST/HTTP 1.1 Success Response Codes

- HTTP successful operations returns one of two codes
  - Code `200 (OK)` with a message body
  - Code `204 (No Content)` without a message body
- REST operations should return 200 (OK) when object is returned
- REST operations should return 204 (No Content) when:
  - The REST method return type is `void`
  - The operation returns a `null` value
    - Not an error condition, informing client of no response body

When a request to a REST/HTTP service is successful a HTTP 200 or 204 status code should be returned. When the REST operation on the server returns a null or the return type is void a HTTP status 204 should be returned.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Setting Response Code in Spring

- `@ResponseStatus` can be used to indicate HTTP status to be returned

```
@GetMapping("/reservation/{id}/confirm")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void confirm(@PathVariable Integer id) {
   …
}
```

```
@PostMapping("/reservations")
@ResponseStatus(HttpStatus.CREATED)
public void makeReservation(@RequestBody ReservationDTO dto) {
   …
}
```

`The @ResponseStatus` annotation can be added to server-side methods to define the return status that should be returned to the client when the method returns (not throwing an exception)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Making the Service RESTful

- Being RESTful is more than just
  - Use URLs to define operations
  - Using the GET, POST, PUT and DELETE method of HTTP
  - Defining CRUD operations for a resource
- In REST the 'engine of application state' must be driven by hypermedia
  - Content contains links to other (related) resources
- Client can dynamically navigate to appropriate resource
  - By traversing hypermedia links
- Client should not require prior knowledge of service or steps that define the workflow
  - Should not have to hard-code URIs of related resources

In the real world, when you visit a website – you hit its homepage. It presents some snapshots and links to other sections of websites. You click on them and then you get more information along with more related links which are relevant to the context.

Similar to a human's interaction with a website, a REST client hits an initial API URI and uses the server-provided links to dynamically discover available actions and access the resources it needs. The client need not have prior knowledge of the service or the different steps involved in a workflow. Additionally, the clients no longer have to hard code the URI structures for different resources. This allows the server to make URI changes as the API evolves without breaking the clients.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Setting the Location Header

- Creating resources typically results in Location header to be set in response
  - Containing URI to newly created resource
- `ResponseEntity` can be constructed using `created` factory method
  - Accepts URI value of Location header

```
@PostMapping("/reservations")
public ResponseEntity<ReservationDTO> makeReservation(
                                          @RequestBody ReservationDTO request) {
  Reservation reservation = service.makeReservation(request.getNameOnReservation());
  URI uri = …
  ReservationDTO dto = modelMapper.map(reservation, ReservationDTO.class);
  return ResponseEntity.created(uri).body(dto);
}
```

When creating a new resource using REST, the response to the client should (beside a HTTP status 201) should include a HTTP header called Location, of which the value is the URI to the newly created resource. To create an instance of ResponseEntity that contains this header value the static factory method ‘created’ can be used.

The question that remains, what would be the best approach for defining the URI

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

UriComponentsBuilder

- Assists in the creation of `URIComponents` (URIs)
  - Including construction, variable expansion and encoding

```
Integer reservationNumber = …
UriComponents uriComponents =
    UriComponentsBuilder.newInstance()
                        .scheme("http")
                        .host("localhost").port(8080)
                        .path("reservations/{reservationNumber}")
                        .buildAndExpand(reservationNumber);
URI uri = uriComponents.toUri();
```

- URLs in REST controller should not be hardcoded

Spring contains the UriComponentsBuilder which assists in the creation of URIComponents (which represents a URI). The builder defines methods for not only defining the different 'parts' of a URI, it also defines methods to support variable expansion (replacing variables in URI string by their actual value) and the character encoding of the URI.

But hardcoding URIs in your code results in code that not only bound to a specific server, it also makes it hard to update and maintain the application

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

ServletUriComponentsBuilder

- Creates links based on current `HttpServletRequest`

```
@PostMapping("/reservations")
public ResponseEntity<ReservationDTO> makeReservation(
                                        @RequestBody ReservationDTO request) {
   Reservation reservation = …
   Integer reservationNumber = reservation.getReservationNumber();
   URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri()
                                        .path("/{id}")
                                        .buildAndExpand(reservationNumber)
                                        .toUri();
   …
}
```

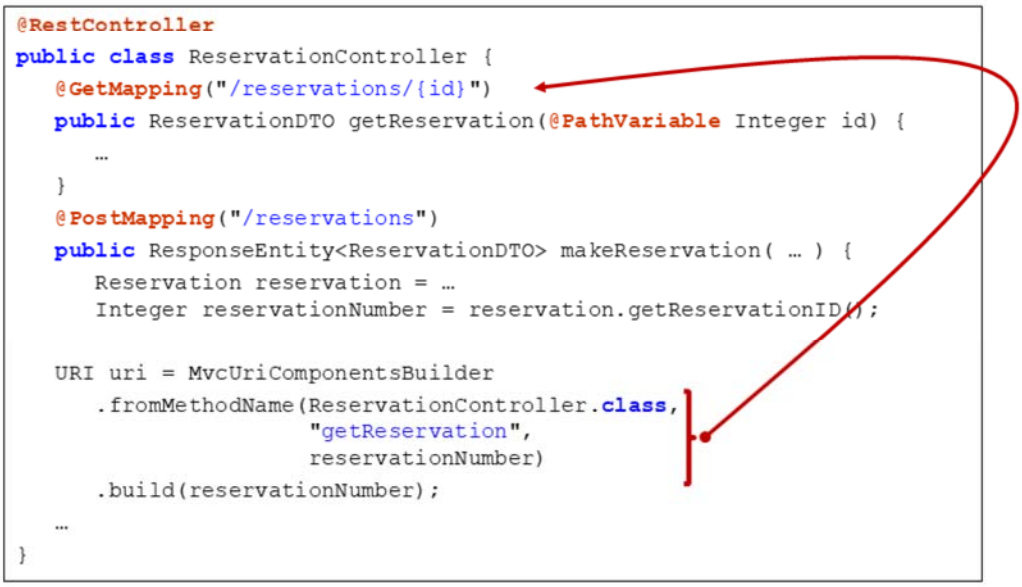
ServletUriComponentsBuilder allows for URIs to be constructed that are based on the current HttpServletRequest. I provides a simple API to create URIs that are relative to the URI that was used to make the current REST call.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

MvcUriComponentsBuilder

- Creates `UriComponentsBuilder` by referencing controller method



MvcUriComponentsBuilder was introduced in Spring 4. It 'obtains' URIs to REST resources by explicitly referencing resource methods on controller classes. The static fromMethodName method accepts the Controller class, the name of the method and the parameters that need to be sent into this method. These parameters are used to determine the parameter types of the method that needs to be invoked on the controller class. MvcUriComponentsBuilder will use this information to determine the URI that has been mapped to the referenced method.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 17: Spring REST Services

~/StudentWork/code/spring-rest-service/lab-code
~/StudentWork/code/spring-rest-services-client/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Session: Spring Security Framework

Enterprise Spring Security
Spring Web Security (JavaConfig)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Enterprise Spring Security

Enterprise Spring Security

Spring Web Security (JavaConfig)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Review basic security concepts as they relate to the Spring framework
- Explore security interceptors and various managers that handle security functions
- Examine options for interacting with various authentication providers
- Understand how authorization through voting works in Spring

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The Main Goals of Security

- Software security has many aspects
  - Control access to resources
  - Integrate with various authentication providers
  - Ensure data is not seen by unauthorized users
  - Ensure services are not accessed by unauthorized users
  - Providing single sign-in capability

  - Ensure users have trust in your system
  - Ensure users have ease of use of system

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Core Security Concepts

- Identity – who do you claim to be
- Authentication – how a user proves that they are who they claim to be
  - Password? Secure-ID? Retina-Scan?
  - Another system that has already authenticated
- Authorization – What are you allowed to do?
- User Details – Information about the person who is currently logged in

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Security Framework

- The Spring Security Framework is a sub-project of Spring
  - Originally called ACEGI
- Providing authentication, authorization and security to enterprise applications
  - For both web and standalone Java applications
- Get basic security with using XML config or Java config files
  - Can connect with existing security systems

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Authentication Models

- Spring Security supports a variety of authentication models
  - HTTP BASIC, Digest, Form, X.509 authentication
  - LDAP and OpenID authentication
  - JA-SIG Central Authentication Service (CAS)
  - Authentication propagation for RMI and HttpInvoker
  - Automatic "remember-me" authentication
  - Anonymous authentication
  - Run-As authentication
  - Java Authentication and Authorization Service (JAAS)
  - Java EE container authentication
  - Java Open Source Single Sign On (JOSSO)
  - ...

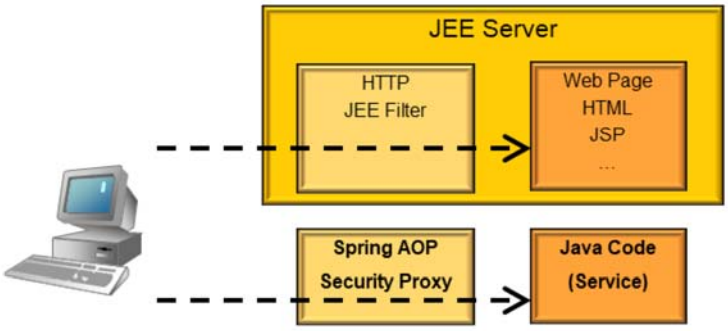
The list continues : AndroMDA, Mule ESB, Direct Web Request (DWR), Grails, Tapestry, JTrac, Jasypt, Roller, OpenNMS Network Management Platform, CA Siteminder

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Security is Transparent to Client

- Client accesses resource the same way it always did
  - Web pages – by URL
  - Java Code – ask Spring for the Java Bean
- Spring Security works by interception
  - Web Pages are protected by HTTP/JEE Filters
  - Service Object are protected by an AOP proxy



Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Authentication Managers

- Determines who you are based on your principal and your credential(s)
  - Credentials are used to prove you are the principal that you are claiming to be
- Pluggable API allows use of most authentication mechanisms, including
  - A database
  - JA-SIG Central Authentication Service (CAS)
    - Supports single sign-on (SSO)
  - LDAP
  - JAAS
  - A remote service
  - x.509 certificates

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

AuthenticationProvider

- Abstraction for fetching user information (e.g. from database, LDAP, ...)
  - Defines `authenticate()` method
  - Takes an `Authentication` object
    - Containing principal and credentials
  - On success, returns populated `Authentication` object
  - On failure, throws `AuthenticationException`

```
public interface Authentication extends Principal, Serializable {
   Collection<? extends GrantedAuthority> getAuthorities();
   Object getCredentials(); //e.g. password
   Object getDetails(); //e.g. IP address, certificate serial number
   Object getPrincipal(); //e.g. username
   boolean isAuthenticated();
   void setAuthenticated(boolean isAuthenticated) throws IllegalArgumentException;
}
```

```
public interface AuthenticationProvider {
   Authentication authenticate(Authentication auth)
                           throws AuthenticationException;
   boolean supports(Class<?> authentication);
}
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Authentication Providers Supplied by Spring

- **Provider classes are located in package**
  `org.springframework.security.authentication`
  - **or subpackages**

Standard Provider implementations	
`JaasAuthenticationProvider`	Uses a configured JAAS authentication service
`RememberMeAuthenticationProvider`	Uses a previously authenticated and remembered user
`RemoteAuthenticationProvider`	Uses a remote service
`TestingAuthenticationProvider`	Used to support unit testing. Treats TestingAuthenticationToken as valid
`X509AuthenticationProvider`	Uses an X.509 certificate
`RunAsImplAuthenticationProvider`	Uses identity substituted by a run-as manager
...	...

```
org.springframework.security.authentication.jaas.JaasAuthenticationProvider
org.springframework.security.authentication.RememberMeAuthenticationProvider
org.springframework.security.authentication.rcp.RemoteAuthenticationProvider
org.springframework.security.authentication.TestingAuthenticationProvider
org.springframework.security.authentication.preauth.x509.X509AuthenticationProvider
org.springframework.security.access.intercept.RunAsImplAuthenticationProvider
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Protecting Authentication Data

- Authentication data must be maintained in some type of persistent repository
- Authentication data often persisted in unencrypted form
  - Configuration files
  - Databases
  - Windows registry
  - Candidate/submitted data
- All persistent storage of authentication data must be obfuscated in some fashion
  - Not sufficient to store in a proprietary, but widely used, format (such as a commercial database)
  - Encryption/decryption may be required

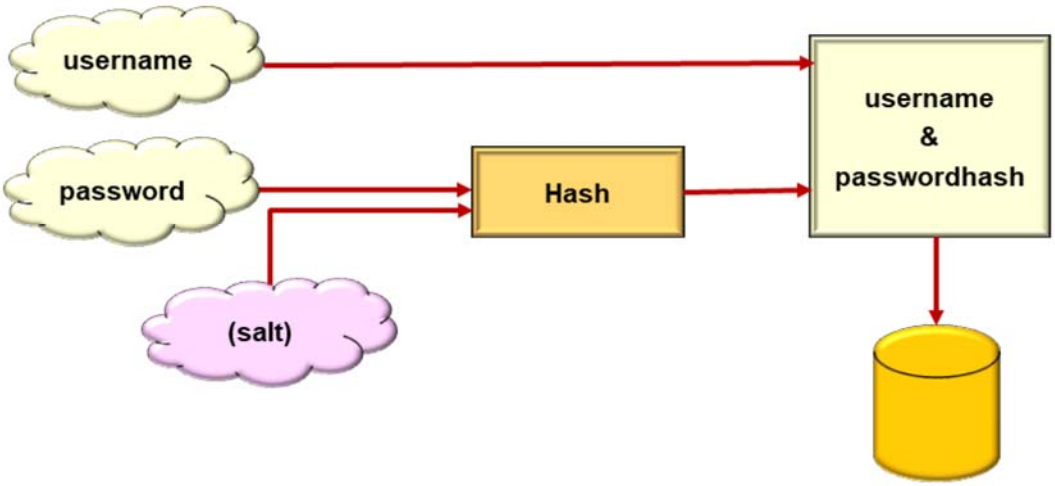
At a high risk due to openness, persistence in untrusted domains, and potential value

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Username/Password Creation Process

- Common encryption method is to use a hash function
  - Hashed values for passwords
    - Need to "salt" the hash process by:
      - Adding a unique string to the original password
      - Unique seed to slightly change the generated hash value

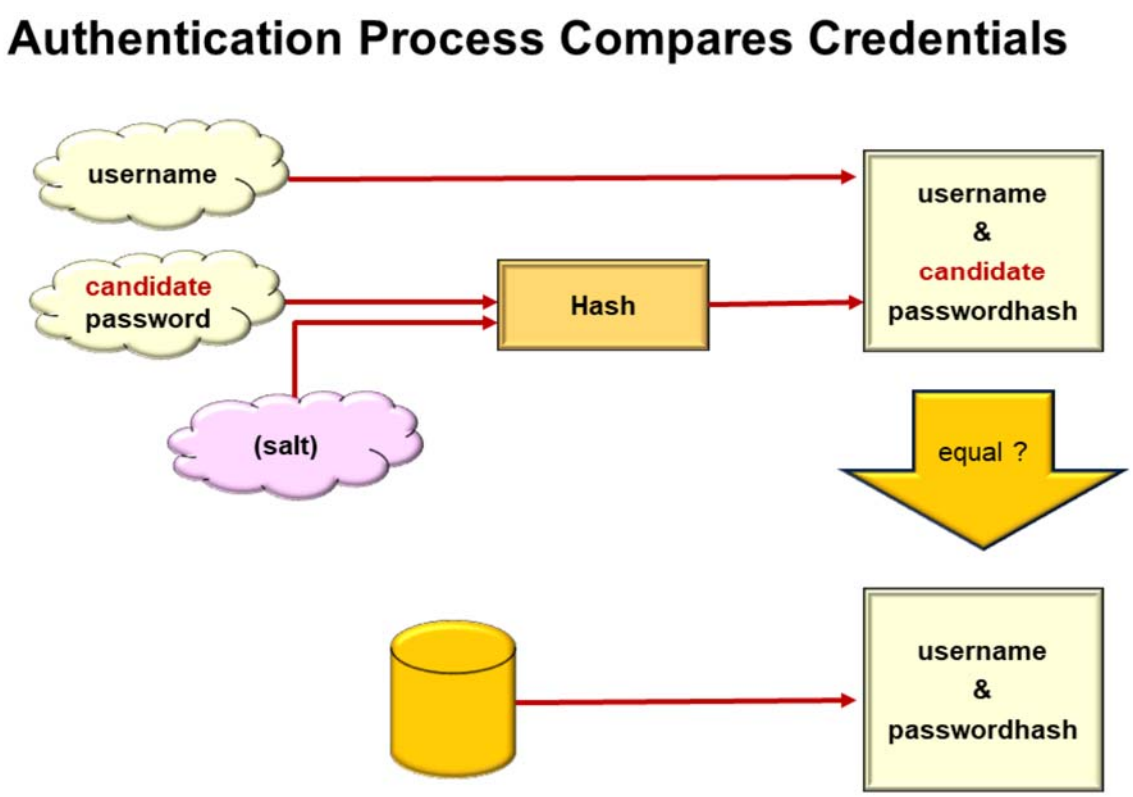


A strong cryptographic hash is characterized by uniqueness (no collisions) and unpredictability (high degree of randomness between the hashed values of two slightly differing initial values).

Hashing algorithms are under a fairly continual (and predicted) process of becoming obsolete and needing to be updated.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.



The hashing process can be supplemented by various salting techniques as well.

Salt and salting techniques: the user sets a password and as a result a short, random string called the “salt” is suffixed to the password before encrypting it. The salt is stored along with the encrypted password so that it can be used during verification. The salt is different for each user therefore the attacker can no longer construct tables with a single encrypted version of each candidate password. This is also referred to as randomizing the hashing process.

Early Unix systems used a 12-bit salt. At that time attackers could build tables with common passwords encrypted with all 4096 possible 12-bit salts. Now, however, if the salt is long enough (e.g. 32 bits) there are too many possibilities and the result is that the attacker must repeat the encryption of every guess for each user.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

DaoAuthenticationProvider

- Interacts with database using Data Access Object (DAO)
  - Assumes that credentials are stored unencrypted
  - Supports use of hashing and salts to reduce exposure via compromise of database
- Can be wired with a hashing component
  - Referred to as a password encoder
    - Available encoders include using Message Digest (MD5), Secure Hash Algorithm (SHA), or LDAP SHA, salted-SHA (SSHA) or BCrypt
  - Default is to use plain text
- Salts can be wired in for the encoder
  - System salt provides same salt for all users
  - Reflection salt is based on a specific user property
    - More secure since each user has a specific salt

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

ProviderManager

- Iterates through list of `AuthenticationProviders`
  - Until `AuthenticationProvider` returns non-null response
    - ..and does not throw `AuthenticationException`.
  - When no capable provider is found `AuthenticationException` is thrown
- Multiple providers can be registered

```
@Autowired
public void configAuthentication(final AuthenticationManagerBuilder auth,
                                 final DataSource dataSource) throws Exception {
       auth.authenticationProvider(customAuthProvider());
       auth.authenticationProvider(jdbcAuthProvider()); //adds JDBC provider
}
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

ProviderManagerBuilder

- `ProviderManagerBuilder` can be used to configure `ProviderManager`

```
@Autowired
public void configAuthentication(final AuthenticationManagerBuilder auth,
                                 final DataSource dataSource) throws Exception {
   JdbcUserDetailsManagerConfigurer<AuthenticationManagerBuilder> jdbcAuth
                                                    = auth.jdbcAuthentication();
   jdbcAuth.passwordEncoder(passwordEncoder())
   jdbcAuth.dataSource(dataSource);
   …
}
```

```
public class JdbcUserDetailsManagerConfigurer<B extends ProviderManagerBuilder<B>>
     extends UserDetailsManagerConfigurer<B, JdbcUserDetailsManagerConfigurer<B>> {
   …
}
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Configuring a DAO Provider (JavaConfig)

```
@Configuration @EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    public void configAuthentication(final AuthenticationManagerBuilder auth,
                                     final DataSource dataSource)
                                                throws Exception {
        auth.jdbcAuthentication()
                .passwordEncoder(bCryptPasswordEncoder())
                .dataSource(dataSource)
                .usersByUsernameQuery(
                  "select username,password, enabled from users where username=?")
                .authoritiesByUsernameQuery(
                         "select username,role from user_roles where username=?");
    }
...
```

```
@Bean
public PasswordEncoder bCryptPasswordEncoder() {
   return new BCryptPasswordEncoder();
}
```

- `SaltSource` does not have to be configured
  - Salt handling is done internally when using BCrypt

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Configuring an In-Memory Provider

- `InMemoryUserDetailsManagerConfigurer` can be used to configure in-memory authentication
- Users can be defined using `withUser` method
  - Returns `UserDetailsBuilder` instance
    - Used to configure password and roles

```
...
 @Autowired
 public void configureGlobalSecurity(final AuthenticationManagerBuilder auth)
                                                                  throws Exception {
  auth.inMemoryAuthentication().passwordEncoder(standardPasswordEncoder());
  auth.inMemoryAuthentication().withUser("user").password("{noop}password")
                               .roles("USER");

  auth.inMemoryAuthentication().withUser("admin").password("{noop}admin")
                               .roles("ADMIN");

  auth.inMemoryAuthentication()
    .withUser("tomcat").password("tomcat")
    .roles("ADMIN", "DBA");//dba have two roles
 }
...
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

GrantedAuthority & ConfigAttribute

- `GrantedAuthority` interface represents authorities granted to a principal

```
auth.inMemoryAuthentication().withUser("Scott")
                             .password("Tiger")
                             .roles("ADMIN");
```

- `ConfigAttribute` interface represents security String
  - May be 'simple' role name
  - Configured using annotations on secured methods

```
@PreAuthorize("hasRole('ADMIN')")
public void create(User user);
```

  - Access attributes for secured URLs

```
http.authorizeRequests().antMatchers("/customers/*").access("hasRole('USER')")
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Access Decision Managers

- Determines whether principal is authorized to use resource being protected
- Combines authenticated information with access control settings for targeted resource
  - Polls one or more voting objects to determine whether access should be granted
    - List of voters included in configuration

```
public interface AccessDecisionManager {
   void decide(Authentication authentication, Object object,
               Collection<ConfigAttribute> configAttributes)
               throws AccessDeniedException, InsufficientAuthenticationException;
   boolean supports(ConfigAttribute attribute);
   boolean supports(Class<?> clazz);
}
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Access Decisions Process

- Manages based on decision process

AccessDecisionManager implementations	
`AffirmativeBased`	Allow access when at least one voter says 'grant'
`ConsensusBased`	Allow access when all non-abstain voters say 'grant'
`UnanimousBased`	Only allows access if all voters say 'grant'

- Access decision voters can vote to allow access, deny access, or abstain from voting
  - Typically, if a vote is abstained it is treated as a denial
- Custom voter can be implemented

```
public interface AccessDecisionVoter<S> {
   int ACCESS_GRANTED = 1; //affirmative
   int ACCESS_ABSTAIN = 0;  //abstain from voting
   int ACCESS_DENIED = -1;  //Negative
   int vote(Authentication authentication, S object,
                Collection<ConfigAttribute> attributes);
   boolean supports(ConfigAttribute attribute);
   boolean supports(Class<?> clazz);
}
```

All implementations shown here are located in the `org.springframework.security.access.vote package`

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

RoleVoter

- One of the standard Spring Security implementations
- Votes when there are one or more authorized roles for a targeted resource
  - Votes when `ConfigAttribute` begins with `ROLE_` prefix
  - Grants access when `GrantedAuthority` returns String equal to one or more `ConfigAttributes`
  - When no `ConfigAttribute` has `ROLE_` prefix, voter will abstain

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

AuthenticatedVoter

- One of the standard Spring Security implementations
- Votes by checking principal for required level of authentication
  - Differentiates between anonymous, fully-authenticated and remember-me authenticated users
- Sites may limit access for remember-me authenticated users
  - Would require users to confirm identity to allow full access

```
@Bean
public AccessDecisionManager decisionManager(){
    UnanimousBased dm = new UnanimousBased(
            new ArrayList(){{
                add(roleVoter());
                add(new AuthenticatedVoter());
            }}
    );
    return dm;
}
```

```
@Bean
public RoleVoter roleVoter(){
    return new RoleVoter();
}
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Run-As and After-Invocation Managers

- Run-As manager can substitute an identity to support further processing
  - Temporarily replace Authentication object
  - Only used in some circumstances, so Run-As managers are optional
- After-Invocation managers checks information being returned for proper access privileges
  - Can prevent value from being returned
  - Can modify returned value to allow it through
  - Only used in some circumstances, so After-Invocation managers are optional

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Web Security (JavaConfig)

Enterprise Spring Security

Spring Web Security (JavaConfig)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Secure Web Pages using JavaConfig
- Extend the `WebSecurityConfigurerAdapter`

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining the Spring Security Configuration

- Configuration can be done using `WebSecurityConfigurerAdapter`

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    …
}
```

- Contains a large set of default settings
  - Requires authentication for all URLs in application
  - Generates a login form
  - Authentication is done using form-based authentication
  - Allows user to logout
  - Defends against CSRF attacks
  - Session Fixation protection
  - …

CSRF – Cross Site Request Forgery

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Defining Spring Security Extensions

- `AuthenticationManagerBuilder` allows definition of `AuthenticationProvider`
  - Multiple authentication-providers can be defined
    - inMemoryAuthentication, jdbcAuthentication, etc.

```
@Autowired
public void configure(AuthenticationManagerBuilder auth)
                                                    throws Exception {
  auth.inMemoryAuthentication()
      .withUser("Fred")
      .password("secret")
      .roles("ADMIN");
   …
}
```

- Method name is not important
- `AuthenticationManagerBuilder` should only be configured in properly annotated classes
  - @EnableWebSecurity, @EnableGlobalMethodSecurity or @EnableGlobalAuthentication

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

HttpSecurity

- `WebSecurityConfigurerAdapter` defines configure method
  - Defining default authentication requirements

```
protected void configure(HttpSecurity http) throws Exception {
   http
      .authorizeRequests()
      .anyRequest().authenticated()
      .and()
      .formLogin()
      .and()
      .httpBasic();
}
```

  - All requests to the application require user to be authenticated
  - Support Form based authentication
  - Configures HTTP Basic authentication

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

HttpSecurity (cont'd)

- `Configure` method is overridden to define access requirements

```
@Override
protected void configure(final HttpSecurity http) throws Exception {
   http.authorizeRequests()
        .antMatchers("/managers/*").access("hasRole('ADMIN')")
        .antMatchers("/**").permitAll()
        .and().formLogin()
        .and().anonymous() //anonymous users represented by role ROLE_ANONYMOUS
        .and().rememberMe() //Allow configuration for Remember me authentication
        .and().logout();  //provide logout support
}
```

- `antMatchers` method defines URL patterns
- `access` method defines access requirements
  - Using EL expression

```
.access("hasRole('ADMIN') and hasRole('DBA')"
```

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Pages Without Access Control

- Not all resources need to be secured

```
.antMatchers("/**").permitAll()
```

  - .antMatchers("/**").anonymous() can be defined
    - Anonymous identity is created
      - Code does not have to check whether UserDetails object is null
    - This access role is not needed – but is used by convention
- Consider allowing easy unsecured access to:
  - Images, CSS
  - Usually, make sure to provide access to index.html

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 18: Using Spring Web Security

~/StudentWork/code/spring-web-security/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Additional Topics: Time Permitting

Introduction to Spring MVC

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Introduction to Spring MVC

Introduction to Spring MVC

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

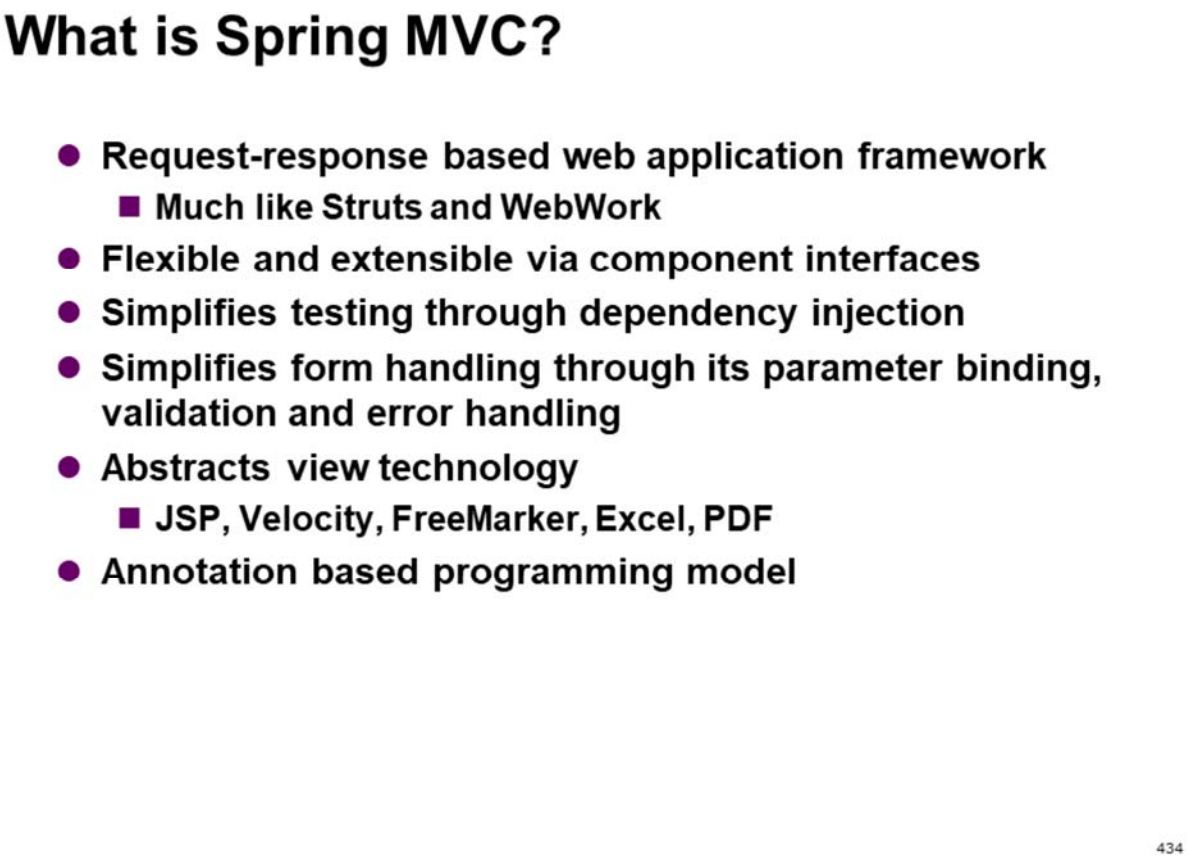
Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Lesson Agenda

- Examine the Spring MVC architecture
- Describe various components in the Spring MVC
- Introduce the `ServletContainerInitializer`
- Configure `DispatcherServlet` in `JavaConfig`
- Use annotations to define Controllers
- Define request handler methods

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.



Spring MVC is a request-response based web application framework providing the foundation for the development of large scale web based applications.

Based on top of the Spring Core APIs, it provides an extremely flexible and extensible framework. By allowing developers to configure every aspect of the request-response cycle, the framework can be altered to support pretty much any web based MVC application.

With the introduction of Spring 3 (and the Servlet 3.0 specification), setting up a Spring MVC application can be accomplished without the need for XML configuration files. Even though XML configuration can still be used to configure all aspects of the framework, Annotations and Java Based configuration have proven to be a more robust approach.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Overview of Spring MVC

- Spring MVC is an MVC web framework
  - Many ready-to-use and highly customizable controllers
  - Many different view models (JSP, PDF, XSL, Excel, etc.) and view resolvers
  - Data binding for forms
  - Wizard functionality for multiple forms
  - Validation
  - I18N internationalization and theme support
  - Interceptors
  - Miscellaneous: file upload, tag library, etc.

Spring MVC is a MVC web framework. Applications built using Spring MVC use several high-customizable controllers to process requests sent by the client, invoke the appropriate business logic and select the view that is to be sent back to the client.

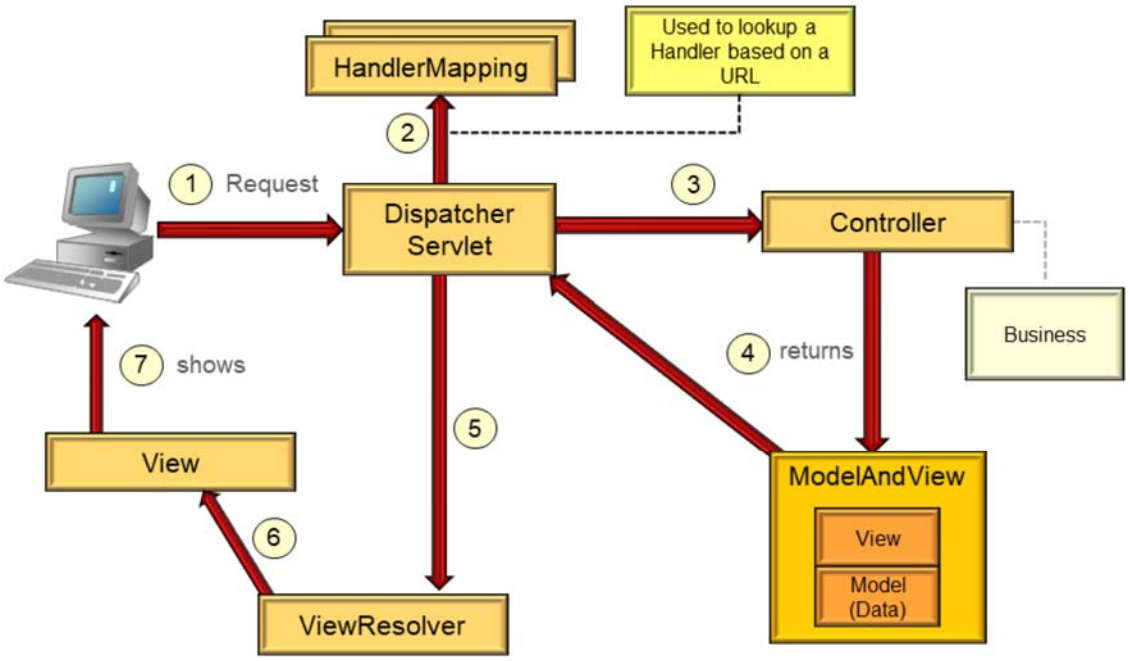
Spring MVC does rely on one particular view technology. The type of response that is sent back to the client depends on the View implementations that are configured for the application.

But Spring MVC is more that just a request-response handling framework. It can assist in all aspects of a large scale web application, ranging from form validation, wizard style implementations, internationalization, file upload and much more.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Request Life Cycle in Spring MVC



The core of the MVC framework is the DispatcherServlet. All requests that are to be processed by Spring MVC must go through the DispatcherServlet. It is this servlet that will determine which controller instance will service the request.

A Controller instance will invoke the business and populate a Model object with the response that is to be shown to the client. Once the controller is done communicating with the business logic, it will return a logical view name and the populated model. This logical view name does not define a specific page or resource that is to be displayed. Instead the DispatcherServlet will rely on ViewResolver instances to map the logical name to a View resource. This View resource can vary from a 'simple' JSP page, a Groovy Markup Template, a PDF document or any other implementation of the View interface.

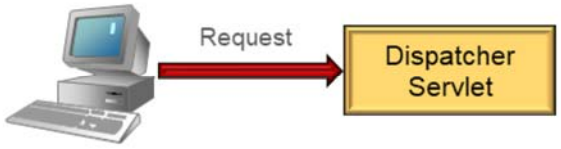
Once the View type has been selected, the response will be rendered using the selected view technology and the information that is available in the model.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

DispatcherServlet

- Main servlet that dispatches to individual controllers
- Entry point for all Spring MVC requests
  - Loads `WebApplicationContext`
  - Controls workflow and mediates between MVC components
    - Dispatches to handlers for processing the web request
  - Loads sensible default components if none are configured



- Example of Front Controller pattern

437

The DispatcherServlet is the main entry point for all requests that are being made to the application. It controls the entire flow of a request-response cycle within the web application. At startup it load an instance of a WebApplicationContext which deals with the web-related components of the application.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

WebApplicationContext

- Spring MVC applications usually have two (or more) `ApplicationContexts`
  - Configured in a parent-child relationship
- Each dispatcher has own `WebApplicationContext`
  - Contains web-specific components
    - e.g. Controllers, ViewResolvers, HandlerMappings, ...
  - Inherits beans defined in root context
    - Contains non-web specific beans like services, DAOs, etc.
- `WebApplicationContext` adds functionality to `ApplicationContext`:
  - Can resolve themes and locales
  - Bound to the `ServletContext` and placed in the request

Spring MVC-based web applications often rely on several ApplicationContext instances. Each DispatcherServlet that is configured within the web application will receive its own WebApplicationContext, while they share a common (parent) ApplicationContext.

The WebApplicationContext contains the web-related components like controllers and view resolvers, while the parent ApplicationContext defines the non-web specific beans, like the service implementation and repositories.

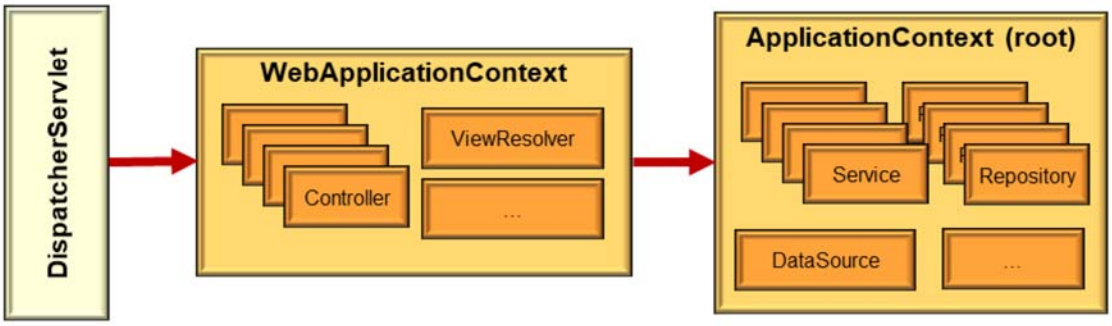
A WebApplicationContext differs from a regular ApplicationContext. Not only is it capable of resolving themes and locales for a request, an instance of the WebApplicationContext is also made available in the request object. (Defined as request attribute DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE)

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring Context Hierarchy

- Spring application context uses hierarchy



- When a context cannot resolve a bean, resolution request will be passed to parent context
  - Bean definitions can be overridden!

Spring uses a hierarchy of application contexts. In a Spring MVC application, the DispatcherServlet receives a WebApplicationContext, which is a child context of a 'regular' ApplicationContext. When a bean instance needs to be resolved by the DispatcherServlet, it will ask the WebApplicationContext to do this. Only when a bean cannot be found it this context will it pass the request to its parent context.

This approach not only allows us to separate the web-based components from the non-web components, it even allows us to 'override' a bean definition in a child context.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Configuring DispatcherServlet: web.xml

- `DispatcherServlet` needs to be configured
  - Needs to be mapped to the URLs it services

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>dispatcher</servlet-name>
   <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

- Handles automatic loading of `WebApplicationContext`
  - Loads context file /WEB-INF/[servlet-name]-servlet.xml
- Context file can be defined using servlet init parameter

```
<init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/dispatcher-config.xml</param-value>
</init-param>
```

The DispatcherServlet must be configured within the JEE web application and one or more URLs needs to be mapped to it.

Each DispatcherServlet that is configured within the application automatically loads its own WebApplicationContext. By default, it will search for an XML file in the WEB-INF folder that has the servlet-names, followed by the –servlet.xml suffix. When the file is name differently or is located in a different location, it can be explicitly specified using the contextConfigLocation servlet init parameter.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The Root WebApplicationContext

- Root context is configured using a ContextLoaderListener

```
<web-app ...version="3.1">
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/services.xml</param-value>
   </context-param>
   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>
   …
</web-app>
```

To load the Root ApplicationContext, a ContextLoaderListener must be configured in the web.xml. The name and the location of the XML configuration file can be defined using the contextConfigLocation context parameter.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Servlet 3.0

- Since Servlet 3.0, web.xml has become optional
  - Programmatic configuration often more convenient
  - Annotations can be used to define configuration
    - e.g. `@WebServlet`, `@WebFilter` and `@WebListener`
- Servlets, listeners and filters can be added programmatically

```
ServletContext container = …
ServletRegistration.Dynamic dispatcher =
      container.addServlet("dispatcher", new DispatcherServlet());
dispatcher.setLoadOnStartup(1);
dispatcher.addMapping("/flights/*");
```

- Initialization is done during startup of web application
  - Using `ServletContainerInitializer` instances

With the release of the Servlet 3.0 specification, the use of the web.xml has become optional. Programmatic configuration has proved to be more convenient and cause less problems at runtime. By using annotations, all configuration that used to be defined in the web.xml can now also be done within the code.

The Servlet API was also extended to allow for programmatic configuration of web resources like servlets, listeners and filters, which made it possible to do all configuration in Java instead of XML.

Instances of ServletContainerInitializer can be supplied to the web application, through which resources can be configured during startup of web application.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Configuring the `DispatcherServlet`

- `DispatcherServlet` can be configured within `WebApplicationInitializer`
  - Providing servlet instance and URL mapping(s)

```
public class WebAppInitializer implements WebApplicationInitializer {
   public void onStartup(ServletContext container) {
      ServletRegistration.Dynamic registration =
              container.addServlet("dispatcher", new DispatcherServlet());
      registration.setLoadOnStartup(1);
      registration.addMapping("/flights/*");
   }
}
```

In order to programmatically register the Spring DispatcherServlet, an implementation of the WebApplicationInitializer must be created and added to the classpath.

The onStartup method of the WebApplicationInitializer takes a single parameter of type ServletContext. Using methods of ServletContext the servlet can be registered and a URL can be mapped.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

AbstractAnnotationConfigDispatcherServletInitializer

- Implementation of `WebApplicationInitializer`
  - Simplifies configuration of dispatcher servlet
  - Registers `DispatcherServlet` instance

```
public class WebAppInitializer
        extends AbstractAnnotationConfigDispatcherServletInitializer {
  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class[] { JavaConfig.class };
  }
  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class[] { MvcConfig.class };
  }
  @Override
  protected String[] getServletMappings() {
    return new String[] { "/" };
  }
}
```

- Recommended approach for configuring Spring MVC

To simplify the configuration of the DispatcherServlet, Spring provides the AbstractAnnotationConfigDispatcherServletInitializer class, which is an implementation of the WebApplicationInitializer interface.

This utility class registers an instance of the DispatcherServlet with the web application. When extending this class, you must provide an implementation of the `getRootConfigClasses`, `getServletConfigClasses and getServletMappings method to define the location of the Spring configuration classes and define the URLs to which the DispatcherServlet must be mapped.`

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Spring MVC Request Beans

- `DispatcherServlet` uses several beans to process request
  - Can be configured in `WebApplicationContext`

Bean	Description
`HandlerMapping`	Defines criteria for which interceptors and controller to use
`Controller`	Defines the request handlers (they contain the process logic)
`ViewResolver`	Maps names to actual View objects (as opposed to being hard coded)
`LocaleResolver`	Resolves the Locale to use for this client (used for I18N)
`ThemeResolver`	Resolves Themes based on input such as Cookies
`MultipartResolver`	Used for file upload capabilities of Spring MVC
`HandlerExceptionResolver`	Maps exceptions to Views

The entire request-response cycle is controlled by the DispatcherServlet. In order to do this, the servlet will make use of a number of beans. For each of these beans, a default has been configured in the WebApplicationContext. To customize the request-response process custom instances of each of these beans can be configured.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@EnableWebMvc

- Enables MVC Java config
  - Should be added to one of the `@Configuration` classes

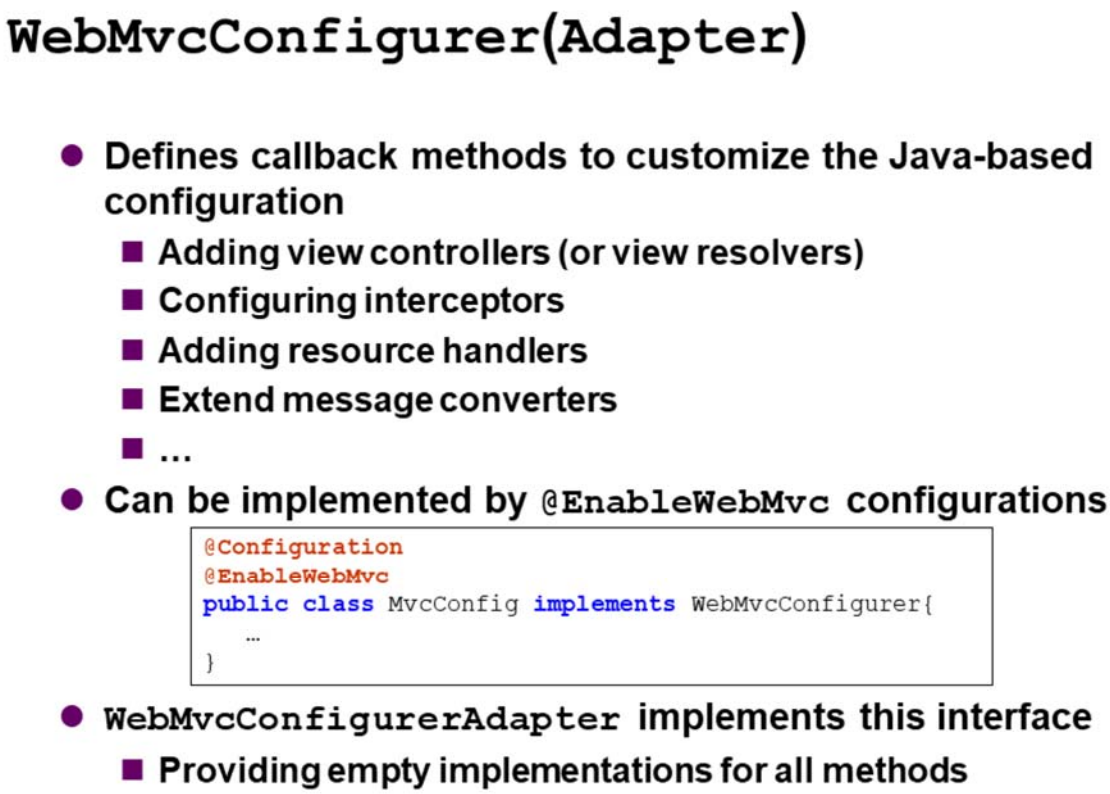
```
@Configuration
@EnableWebMvc
public class MvcConfig{
    …
}
```

- Registers handlers for request mapping annotations
  - `@RequestMapping`, `@GetMapping`, …
- Adds support for number & date formatting annotations
  - `@NumberFormat`, `@DateTimeFormat`
- Adds support for bean validation
  - Bean validation API must be on classpath
- Adds support request body method parameters
  - String converters, XML converters, etc…

To make full use of the annotation and JavaConfig configuration, one of the @Configuration classes should be annotated using the @EnableWebMvc annotation. Not only does this register a HandlerMapping to support annotation based configuration of controllers, it also enables the use of several other annotations like number and date formatting and validation.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.



Once the servlet has been configured within the web application, the WebApplicationContext for this dispatcher can be configured. Even though most of the default configuration is sufficient for a simple web application, to make use of the more advanced features of Spring MVC, changes to the configuration must be made.

To simplify the customization of the Spring MVC runtime, a `WebMvcConfigurer interface defines a set of callback methods. By implementing one or more of these methods, beans like ViewResolvers and interceptors can be registered with the context.`

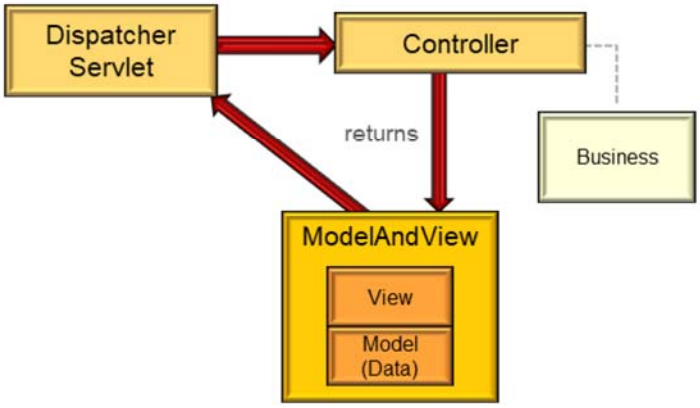
`WebMvcConfigurerAdapter is an implementation of the WebMvcConfigurer, which defines empty implementations of all the methods defined by the interface. By extending this class instead of implementing the interface, you only have to override the methods you are interested in.`

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Controllers

- Provide access to application behavior (services)
  - Interpret user input
  - Map input to application model
  - Invoke service
  - Select view to represent result
- Controllers can be defined using annotations
  - `@Controller`
  - `@RequestMapping`
  - `@RequestParam`
  - ...



In early versions of Spring MVC, controller classes had to implement the Controller interface. Most implementations extended one of the available base class implementations such as `AbstractController`, `SimpleFormController`, `MultiActionController`, or `AbstractWizardFormController`.

`Since Spring 3, annotations are the preferred way to define the controller class and the methods responsible for handling the request.`

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@Controller Annotation

- Indicates that a class serves the role of a controller
  - No need to extend controller base class or reference Servlet API
- @Controller annotation is a stereotype annotation
  - Will be discovered when classpath scanning is enabled

```
@Controller
public class FlightController {
    …
}
```

- Dispatcher will scan class for mapped methods
  - Methods that service requests

By annotating a class using the @Controller annotation, we are indicating that the class service with the role of controller within the web application. The class does not have to implement a specific interface or extend a specific class anymore (like was the case in earlier versions of Spring MVC)

The @Controller annotation is a stereotype annotation. Not only does it identify the class as a controller, it also makes is available for automatic discovery when classpath scanning is enabled within the context.

The actual processing of the HTTP request is done by a (mapped) method defined within the class. A single controller class may contain several methods capable of handling requests. Annotations added to the methods define which method is invoked for which request.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@RequestMapping

- Maps request URLs to controller and/or methods
  - Used at both the class and method level

```
@Controller
@RequestMapping(value = "/flights")
public class FlightController {
   @RequestMapping(path = "allFlights", method = RequestMethod.GET)
   public String getAllFlights() {
      …
   }
}
```

- Annotation at class level maps URL to controller
  - Annotation at method level narrows class mapping

http://localhost:8080/SpringWeb/flights/allFlights GET request

- When only defined at method level, path is absolute URL
- Request handling methods can have flexible signatures
  - Variety of parameter and return types are supported

To map a URL to a controller (method), the RequestMapping annotation is used. When added to the class, the URL it defines is the base URL for all mapped methods defined within that class. The RequestMapping at method level then narrows the mapping by providing more information about the request.

When the class does not contain a RequestMapping annotation, all URLs defined at method level are absolute URLs.

The actual method signature of the mapped method is pretty flexible. Later, we will introduce some of the parameter and return types that can be used.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@RequestMapping (cont'd)

- **Annotation attributes can be used to narrow mapping**

Element	Description	Type	Default
name	Name of the mapping	String	""
path / value	(URL) Mapping	String[]	{}
method	Http request method type	RequestMethod[]	{}
params	Parameters of the request	String[]	{}
headers	Headers of the request	String[]	{}
consumes	Media types of the request	String[]	{}
produces	Produced media types	String[]	{}

```
// http://localhost:8080/SpringWeb/flights/allFlights?overview=true
@RequestMapping(path = "allFlights", params = "overview=true")
public String getOverviewOfFlights() {
   …
}
//http://localhost:8080/SpringWeb/flights/allFlights
@RequestMapping(path = "allFlights")
public String getAllFlights() {
   …
}
```

The mapping of a request to a method is not limited to the URL that was used to make the request. Several attributes of the RequestMapping annotation allow for more details of the request to be defined. Not only can the HTTP type of the request be defined (e.g. GET / POST) the mapping can also be narrowed by defined request parameters that must be present in the request or header information available in the request.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Composed Mapping Variants

- Spring 4.3 introduced method-level mapping annotations
  - Alternative to the `@RequestMapping` annotation
- Simplifies mapping of HTTP methods
  - `@GetMapping`
  - `@PostMapping`
  - `@PutMapping`
  - `@DeleteMapping`
  - `@PatchMapping`

Element	Description
name	Name of the mapping
path / value	(URL) Mapping
params	Parameters of the request
headers	Headers of the request
consumes	Media types of the request
produces	Produced media types

```
//Same as @RequestMapping(path = "allFlights",
//                        method = RequestMethod.GET)
@GetMapping(path="allFlights")
public String getAllFlights() {
   …
}
```

Spring 4.3 added several method-level annotations as an alternative to the RequestMapping annotation. For each of the HTTP request methods, a separate annotation was defined. So instead of using the method attribute of the RequestMapping annotation, one of these annotations can be used.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Handler Method Parameters

- Supported method parameter types include
  - Request and Response object defined by Servlet API
    - ServletRequest, HttpServletRequest, ...

```
@GetMapping(path = "allFlights")
public String getAllFlights(HttpServletRequest request) {   …   }
```

  - Session object (defined by Servlet API)
    - HttpSession
  - Map implementations representing the model
    - java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap

```
@GetMapping(path = "allFlights")
public String getAllFlights(Model model) {   …   }
```

  - Types for accessing request and response content directly
    - InputStream, OutputStream, Reader, Writer
  - Several other (annotated) parameter types

The handler method can be defined to accept a variety of parameters. Parameters of type ServletRequest and ServletResponse can be defined when the implementation needs direct access to the request object. An instance of HttpSession can be provided as a method parameter when access to the user's session is required.

A parameter of type Map, Model or ModelMap can be defined when the implementation needs access to the model for this request, allowing implementations to add objects to the model. Other parameters include InputStream and OutputStream (or Reader and Writer) types which provide direct access to the request (and response) body.

Several other parameter types can also be used, but additional parameter-level annotations need to be provided in order for these parameters to be set.

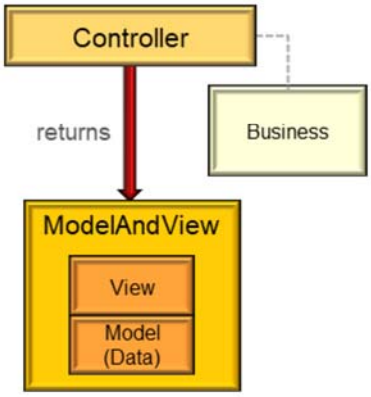
Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The Model

- Supplies attributes used when rendering view
  - Data to be displayed is added to model

```
@GetMapping(path = "allFlights")
public String getAllFlights(Model model) {
   List<Flight> allFlights = …
   model.addAttribute("searchResult", allFlights);
   …
}
```



The Model object must contain the objects (attributes) that are eventually going to be used when rending the view. Attributes added to the model are added under a unique key. This key is later used by the view to retrieve the object from the model.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

ModelMap

- `ModelMap` allows object to be added without specifying a name
  - Naming convention is used to define name of object
    - Using the short class name of the object's class
    - Arrays are named after the Array type, suffixed with 'List'
    - For Set and Lists, type of first element in collection is used (suffixed with 'List')
    - A HashMap will be added under the name 'hashMap'

```
@GetMapping(path = "getReservation")
public String getReservation(ModelMap model) {
   Reservation reservation = …
   List<Flight> flights = …
   model.addAttribute(reservation); // reservation
   model.addAttribute(flights); //flightList
   …
}
```

    - Adding null pointer results in `IllegalArgumentException`

A ModelMap is a specialized Model implementation that does not require the explicit definition of an object name when adding objects to the model. When adding an object to the model, the short class name of the object will be used and the name of the object. When the object added is of type Array, it will use the array type suffixed with 'List'. When the object type being added is of type Set or List, the implementation will look at the first element in the collection and use the short class name of that element suffixed with 'List'.

When adding a HashMap, the name used by the implementation is 'hashMap'. Since this is not the most intuitive naming of object, it is recommended to explicitly define a name when adding an object of this type.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

@RequestParam

- Request parameters can be bound to method parameters

```
@GetMapping(path = "allFlights")
public String getAllFlights(@RequestParam(name = "airline") String airline) {
```

- By default parameter is required
  - Can be made optional by setting required element to false
    - (Optionally) defining default value

```
@RequestParam(name = "airline", required = false, defaultValue = "KLM")
```

- Type conversions are automatically done

```
public String getAllFlights(
    @RequestParam(name = "maxResults") int maxResults){
```

- When added to `Map<String,String>` map is populated with all request parameters

```
public String getAllFlights(
    @RequestParam Map<String, String> requestParameters, ModelMap model)
```

Responding to an HTTP request often requires the interpretation of the request parameters that were sent along with the request. The @RequestParam annotation can be used to annotate parameters of the mapped method. By default, all parameters annotated with this annotated are required, an exception is thrown when the parameter is not available in the request. When the parameter might not be present in the request the required attribute of the annotation can be set to false and an (optional) default value can be specified.

When the parameter is not of type String, the implementation will attempt to convert the request parameter to the appropriate type. By default all ‘simple’ types are supported (int, long, Date, etc.). When additional data types need to supported, a WebDataBinder can be registered or Formatters can be registered with the FormattingConversionService.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

WebDataBinder and @InitBinder

- Request parameter binding can be customized
  - Through the use of the `WebDataBinder`
- `@InitBinder` annotated methods can be added to the controller class
  - Method may not return a value
  - Parameters include
    - `WebDataBinder, WebRequest and Locale`

```
@Controller
public class FlightController {
   @InitBinder
   public void init(WebDataBinder dataBinder) {
      dataBinder.addCustomFormatter(new LocalDateFormatter());
   }
   …
}
```

```
@GetMapping(path = "/getFlights")
public String getAllFlights(
   @RequestParam(name = "destinationCode") String destinationCode,
   @RequestParam(name = "departureDate") LocalDate departureDate,
    ModelMap model) {   …   }
```

When the data type of a parameter is not one of the supported ‘simple’ types or the formatting is specific within the application (for example a date pattern), a custom binding can be defined. To register a specific binding within the controller class, a method can be annotated using the @InitBinder annotation. This method should not return a value, but can accept a number of parameters including most of the parameter types that are also allowed by @RequestMapping methods.

Most often the binder method accepts at least a parameter of type WebDataBinder, which allows for the registration of custom formatter implementations

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

WebDataBinder and @InitBinder (cont'd)

- Allows registration of `Formatter` or `PropertyEditor`

```
public class LocalDateFormatter implements Formatter<LocalDate> {
   public static final DateTimeFormatter FORMATTER =
                            DateTimeFormatter.ofPattern("dd/M/yyyy");
  @Override
  public String print(LocalDate object, Locale locale) {
    return object.format(FORMATTER);
  }
  @Override
  public LocalDate parse(String text, Locale locale) throws ParseException {
      return LocalDate.parse(text, FORMATTER);
  }
}
```

- Automatically bound to generic type of formatter

```
dataBinder.addCustomFormatter(formatter);
```

  - ...or explicitly to a type

```
dataBinder.addCustomFormatter(formatter, LocalDate.class);
```

The actual formatting logic that is registered is implemented in a class that implements either the Formatter or the PropertyEditor interface. (Since Spring 4.2 Formatter implementations are preferred).

The generic Formatter interface define two method, print and parse, in which the conversion to and from a String value must be defined.

When the formatter is registered with the WebDataBinder, formatter is automatically applied to any parameter that is of the same type as the generic type of the formatter. When a more generic formatter was defined, the parameter type to which the formatter is to be applied can also be defined explicitly.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Handler Method Return Type

- Supported method return types include:
  - `ModelAndView` : Model and view information
  - `Model` : Model with implicit view name
  - `Map` : Model with implicit view name
  - `View` : Specific implementation for rendering content
  - `String` : Logical view name
  - `void` : Response is handled within method
  - ...
- Handler methods should return logical view name
  - View Resolver will select View

The mapped handler method may define a number of different return types. When the method returns void, it should take care of rendering the result itself. In other cases, the method should at least return the logical view name of the view that is to be presented to the client. Additionally, it might return the model, containing the data that is to be used when generating the view.

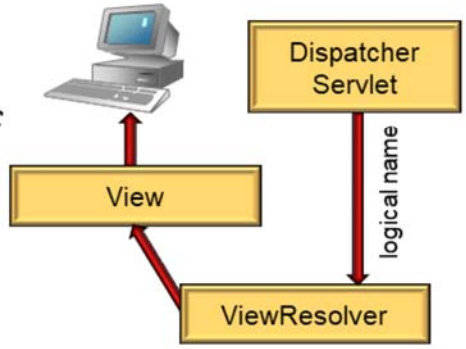
The logical view name that is returned by the method will be used by a ViewResolver instance to determine the actual view that is to be presented to the client.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

ViewResolver

- Resolves a logical view name to a `View` object
  - Multiple resolvers can be defined (in specific order)
- Typical use is `InternalResourceViewResolver`
  - In applications using JavaServer Pages
- Other `ViewResolver` implementations
  - `VelocityViewResolver`
  - `FreeMarkerViewResolver`
  - `ResourceBundleViewResolver`
  - `XmlViewResolver`



A View resolver is responsible for using the logical view name to select the actual view that is to be rendered. A typical use is the InternalResourceViewResolver, which uses the logical view name to lookup the JSP page that needs to be presented.

Spring MVC does come with a large number of alternative ViewResolver implementations, which allows for alternative view technologies to be used to render the response.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

InternalResourceViewResolver

- Default `ViewResolver`
  - Used unless explicit `ViewResolver` is configured
    - Registered using `@Bean`

```
@Bean
public InternalResourceViewResolver viewResolver() {
   InternalResourceViewResolver resolver = new InternalResourceViewResolver();
   resolver.setPrefix("/WEB-INF/views");
   resolver.setSuffix(".jsp");
   return resolver;
}
```

    - Defined using `WebMvcConfigurerAdapter`

```
@Configuration
@EnableWebMvc
public class MvcConfig extends WebMvcConfigurerAdapter {
   @Override
   public void configureViewResolvers(final ViewResolverRegistry registry) {
      registry.jsp("/WEB-INF/views/", ".jsp");
   }
}
```

The `InternalResourceViewResolver`is the default View Resolver in Spring MVC. By default, it will use the URI that was provided (the logical view name) to resolve the resource that is to be presented.

The resolver can be customized by defining a custom implementation either as a bean within the WebApplicationContext, or by registering it by overriding the configureViewResolvers method of the WebMvcConfigurerAdapter

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

InternalResourceViewResolver (Cont'd)

- Uses View class for all generated views
  - By default uses `InternalResourceView`
  - Uses `JstlView` when JSTL is present on classpath
  - Can be specified using `setViewClass` method
- When defining multiple ViewResolvers, must be last in list
- JSPs are often placed under WEB-INF folder
  - Making them only accessible to controllers

```
@Override
public void configureViewResolvers(final ViewResolverRegistry registry) {
   registry.jsp("/WEB-INF/views/", ".jsp");
}
```

By default, the `InternalResourceViewResolver uses an InternalResourceView as the type of view that is to be presented. However, when the JSTL libraries are available on the classpath, it will automatically switch to the JstlView implementation. When a different View implementation is required, an instance of View can be specified by invoking the setViewClass on the InternalResourceViewResolver.`

A Spring MVC application can define multiple ViewResolver instances. The type of view that is to be sent back to the client might depend on information within the request (parameters, request headers, URL). When defining multiple ViewResolver instances, we must make sure that the `InternalResourceViewResolver is always the last one in the list. InternalResourceViewResolver a`ttempts to resolve any view name, even when the underlying resource does not exist (resulting in a 404 error code).

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The View

- View technology is separated from MVC Framework
  - View technology to be used can be configured
- Provides support for a number of view technologies
  - JavaServer Pages (JSP and JSTL)
  - Groovy Markup Template Engine
  - Thymeleaf, Velocity, Freemarker, ...
- Rendering response as XML
  - Tiles
  - XSLT
- Rendering document Views
  - PDF
  - Excel

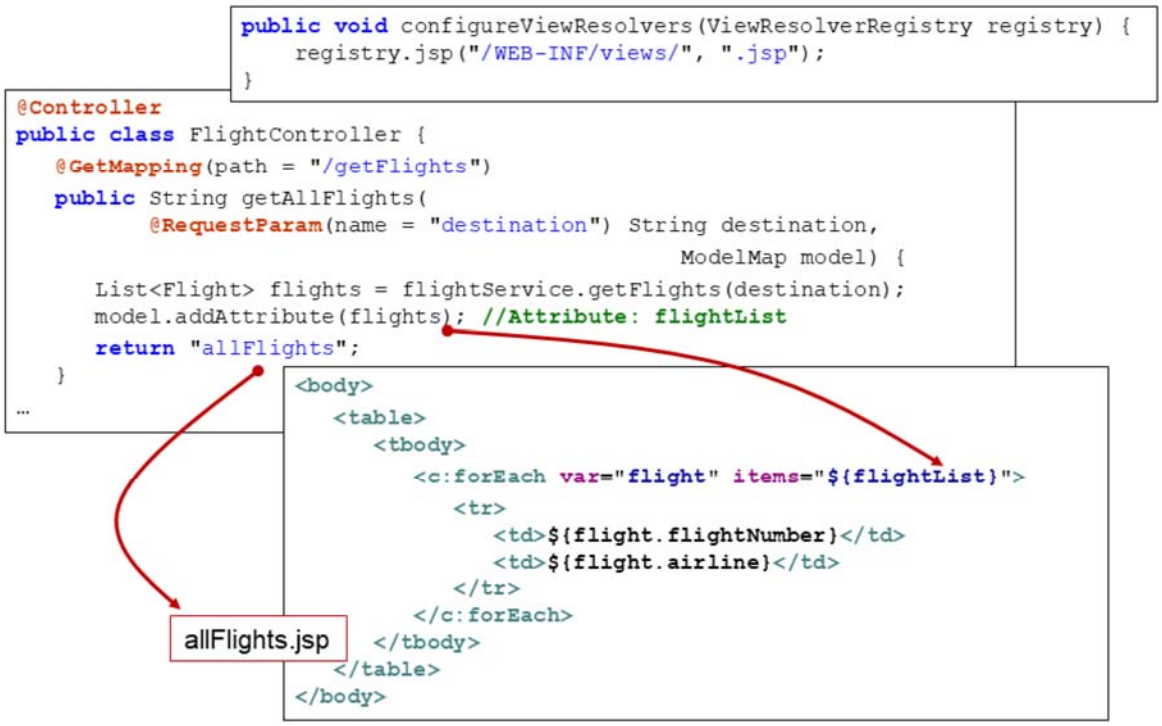
One advantage of the Spring MVC framework is that the View technology that is used to generate the response is complete separate from the framework. Different View technologies can be configured and used by the dispatcher. Out of the box, Spring does come with a number of View technologies that it supports. Not only does it provide support for JSP and JSTL, defining the view using the Groovy Markup Template Engine, Thymeleaf, Velocity of Freemarker is also possible.

Spring even provides support for rendering the response as XML, relying on the Tiles or XSLT frameworks. To create output in PDF or Excel format, Spring MVC defines abstract View classes that can be used as base class for the creation of these types of documents.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

The JSP/JSTL View



When using the JSP/JSTL View(Resolver), the logical view name that is returned by the controller will be used to locate the appropriate JSP class. The JSP class will make use of JSTL tags and the EL expression language to retrieve information from the Model and render the response.

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Exercise 19: Using Spring MVC

~/StudentWork/code/spring-mvc/lab-code

Please refer to the written lab exercise and follow the directions as provided by your instructor

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Thank you for attending this course.

Please remember to turn in your completed course evaluations

Questions? Please contact Training@triveratech.com

Copyright © 2019 Trivera Technologies LLC., Worldwide. All rights reserved.

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.

Trivera Technologies LLC - Worldwide | Educate. Collaborate. Accelerate!
Global Developer Education, Courseware & Consulting Services

JAVA | JEE | OOAD | UML | XML | Web Services | SOA | Struts | JSF | Hibernate | Spring | Admin

IBM WebSphere | Rational | Oracle WebLogic | JBoss | TomCat | Apache | Linux | Perl

609.953.1515 direct | Training@triveratech.com | www.triveratech.com

Course Material for the GK-JPMC TT3325 Core Spring course the week of Oct. 28. 2019. Not for any other use or distribution.