

Pro Dev Session



Pro Dev Session

You may want to write this down.

Collaboration is an essential part of being a great teammate and developer. To be a good collaborator, we must master our collaboration tools, namely git and GitHub.

Today we will get more practice with git. We'll be building on this for the next several lessons.



Pro Dev Session

You may want to write this down.

Recall what we've learned so far ...





Check-in Time

- What git command do you use to temporarily save changes that you don't want a permanent record of?
- What git command is used to permanently save files exactly as they are in that moment?
- What git command tracks (or stages) all files in a project?
- What git command is used to create a safe isolated place to work on new features?
- What git command is used to see a list of all previous saved versions?
- What git command is used to see a specific older saved version of the project?



Git Demo

WATCH & LEARN

Close your laptop. Eyes on my screen. Pay attention.

Pay careful attention to this demo of:

```
git clone
git add -A
git commit -m "<message here>"
git status
git stash
git branch <branchname>
git log
git checkout <branchname or SHA>
```

Pro Dev Session

You may want to write this down.

Today we will get more practice with these commands before we add new ones in the next class.



Git Practice

Work individually but with the support of your classmates to complete all of the goals below.

Goals:

- Create a new repository on GitHub
- Clone the repository to your local machine
- Create a new IntelliJ project with a main method that prints Hello World
- Add, commit, and push your changes to master
- Create a new branch called math and switch to this branch
- Use the `status` command to validate that you are on the correct branch.
- Create an add, subtract, multiply, divide, and exponentiate method.
- Overload at least two of the method.
- Add and commit these changes.
- Return to the master branch. Create a new branch called concat and switch to it.
- Add a method that concatenates multiple strings. Overload this method.
- Use `git pull origin master` to ensure your branch is up to date with the master on GitHub.
- Add, commit, and push your changes using `git push origin concat`.
- On github, create a pull request and merge it.
- Locally, switch to your master branch and use `pull` to update it. With the changes.
- Switch to your math branch. Determine the best way to create a pull request and merge it into master.

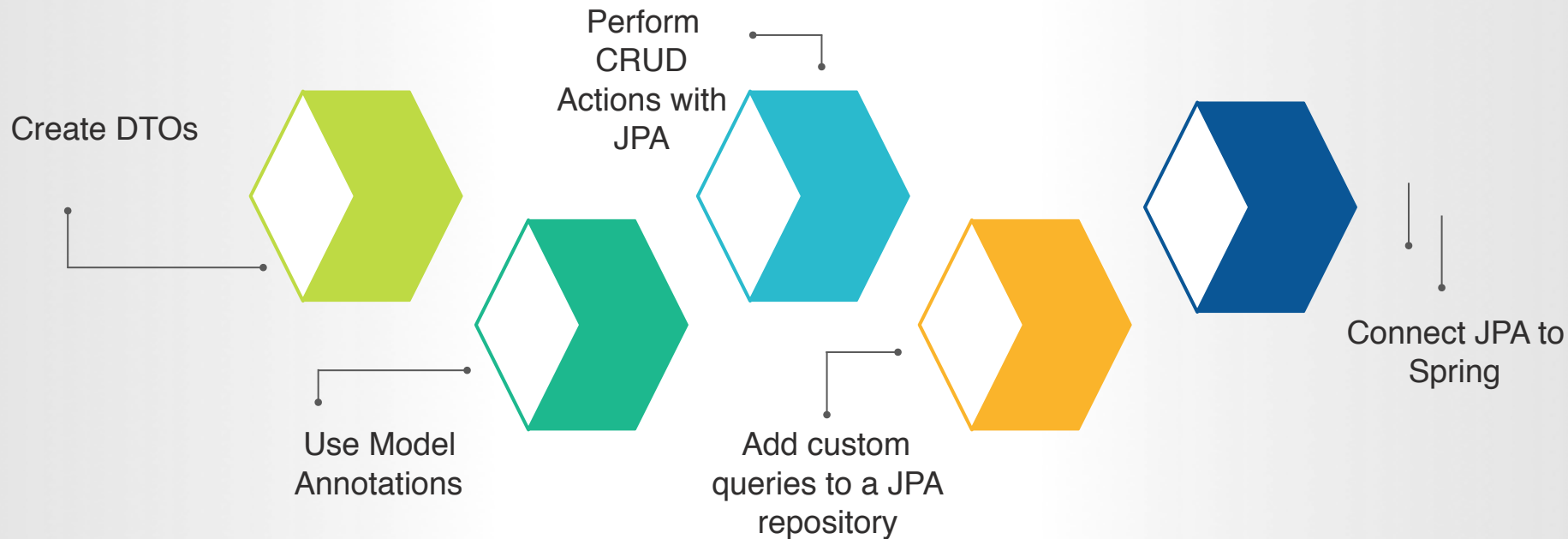
Stand Up!



Objectives & Key Outcomes

THE TAKEAWAYS FROM THIS CLASS

By the end of class today, you will be able to:



Objectives & Key Outcomes

THE TAKEAWAYS FROM THIS CLASS

By the end of class today, you will be able to:



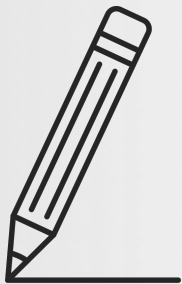
Let's Do This!



DTOs

DTO

SQL ENTITY TO JAVA OBJECT



Notebooks Ready? It's time for a brief lecture.



DTO

SQL ENTITY TO JAVA OBJECT

SQL databases don't speak Java and Java doesn't speak SQL.

A Java application has no idea what a table is or a row or a column. Java doesn't have rows and columns, it has objects and properties.

In a table called Customer with the columns id, name, and address, what do you think we'd call our class? What attributes would the class have?



DTO

SQL ENTITY TO JAVA OBJECT

In order to pull SQL data into Java, we have to create objects to represent each row in a table. To create objects, we first must create a class.

You'll sometimes hear this class referred to as an **Entity** or **Model** occasionally, though those can also have other meanings.

We'll be calling this class by its more precise name -- DTO or **Data Transfer Object**.



DTO

INDEPENDENT PRACTICE

It's time to fly. Focus. Work hard. Ask for help when you need it.

Work in PAIRS to complete all of the goals below.

Goals:

- Create a class called Note
- Include properties for every field in the SQL table. Be sure to use the appropriate type for each.
- Implement getters and setters for the fields.

SQL Table:

```
CREATE TABLE IF NOT EXISTS note (  
    id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    Content TEXT NOT NULL,  
    Customer_id INTEGER NOT NULL  
);
```



**10
minutes!**

Model Annotations

MAPPING DTO TO DATABASE SCHEMA

Spring Data **JPA** takes advantage of object relational mapping (**ORM**) technology.

- This technology automatically maps Java objects to database tables.
- The framework that we'll be using is called **Hibernate**, but Spring Data JPA abstracts the details of the underlying ORM technology away for us so we don't have to worry about the details of Hibernate.

To use JPA, we'll use **annotations** in our DTO to provide the mapping from Java property to database field. But first, we need to include JPA in our dependencies.

Go to <https://start.spring.io> and create a new project. This time, in addition to **Web**, add **JPA** as one of the project dependencies.



Model Annotations

WATCH & LEARN

Close your laptop. Eyes on my screen. Pay attention.

```
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
@Table(name="customer")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;
    private String firstName;
    private String lastName;

    private String company;
    private String phone;

    @OneToMany(mappedBy = "customerId", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<Note> notes;

    // Getters and setters omitted for brevity
}
```

Model Annotations

MAPPING DTO TO DATABASE SCHEMA

@Entity — class-level annotation that marks this as a persistable object. The class maps to a table.

@Table — class-level annotation that specifies the name of the table this class maps to. This is optional. The table name will be the class name if this annotation is not present.

@Id — property-level annotation that indicates that this is the primary key/identifier for this class.

@GeneratedValue — property-level annotation that indicates that the value for the property is generated.

- We use `strategy=GenerationType.AUTO` because we want the database to auto-generate this value for us. This maps to `auto_increment` in MySQL.

@OneToMany — property-level annotation that indicates that this is the one side of a one to many relationship.

- The `mappedBy` attribute indicates the property in the associated object that acts as the foreign key (in our case, it is the `customerId` property of the Note class).

@JsonIgnoreProperties — class-level annotation that specifies properties that should be ignored when serializing this object to JSON.

Model Annotations

MAPPING DTO TO DATABASE SCHEMA

When creating DTOs for use with JPA, we use the **wrapper class** instead of primitives. We do this because in some instances, fields can be **null** in a SQL database. Since primitives can't be null, we use the wrapper class.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Integer id;
```

Model Annotations

INDEPENDENT PRACTICE

It's time to fly. Focus. Work hard. Ask for help when you need it.

Work in PAIRS to complete all of the goals below.

Goals:

- Add all the requisite annotations to your DTO from the previous exercise.

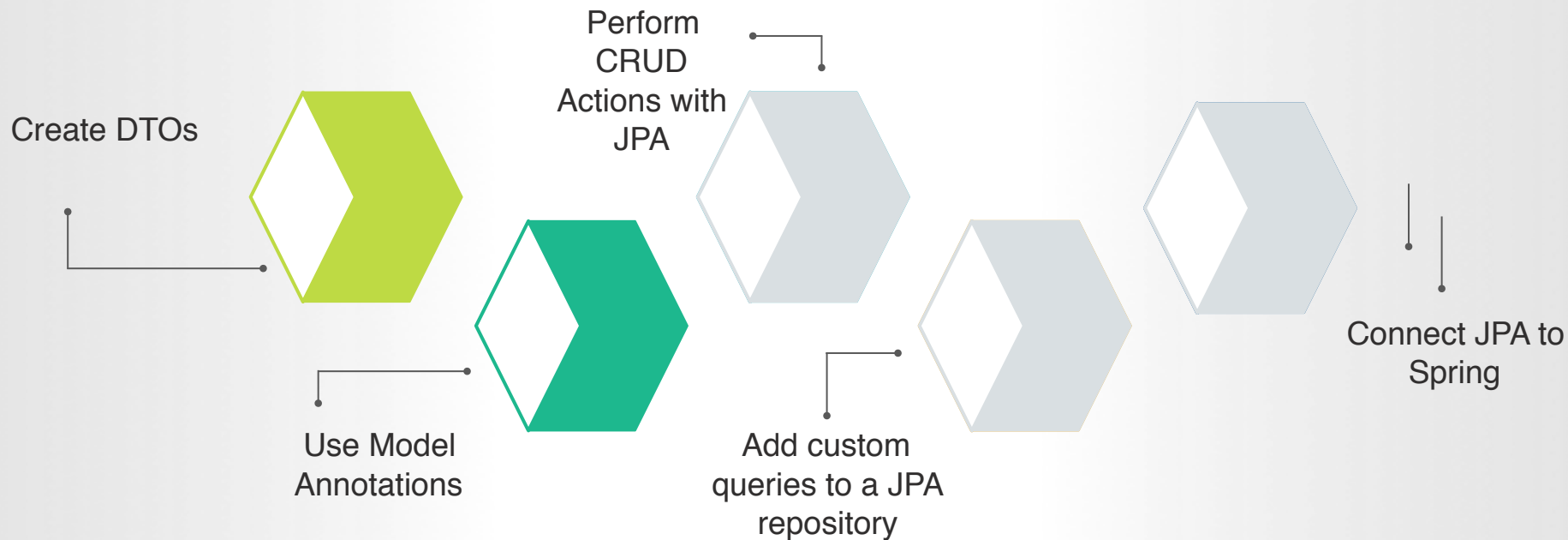


**10
minutes!**

Objectives & Key Outcomes

THE TAKEAWAYS FROM THIS CLASS

By the end of class today, you will be able to:





Stay Seated & Take 3 Deep Breaths.

RELAX.

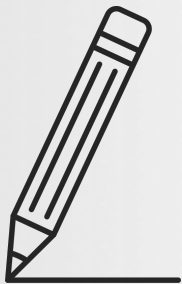
Now take a short walk. Clear your head. After a few minutes break, quickly review your notes.
We'll start back shortly.



JPA

JPA

JAVA PERSISTENCE API



Notebooks Ready? It's time for a brief lecture.



JPA

JAVA PERSISTENCE API

Now that we know how to create a class that models the entities in our database, let's connect to our database directly from our Java application. In the `resources` directory, save the following in the `application.properties` file.

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

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true
```

This creates a database schema for us. No more typing out SQL by hand.



JPA

JAVA PERSISTENCE API

If we want to work with an existing schema, we alter the code slightly.

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

spring.jpa.hibernate.ddl-auto=none

spring.jpa.show-sql=true
```



Warning: JPA feels like magic. That's because some awesome team of developers went in and did all the heavy implementation work for you.

This is an awesome example of why abstraction and encapsulation are so powerful!



JPA

JAVA PERSISTENCE API

JPA's job is to get data from SQL and pull it into a Java application, creating objects from our DTO to represent each row.



JPA

JAVA PERSISTENCE API

Let's checkout the [JavaDoc page](#) to see what methods we get for free when we use JPA.

count()

delete()

deleteAll()

deleteById()

existsById()

findAll()

findAllById()

save()

saveAll()



JPA


JAVA PERSISTENCE API

With JPA, we can simply define an interface and use the `@Repository` annotation to use all these awesome methods to get our data from SQL to our Java Application.

We don't even have to implement the interface. We just need to define it, add the annotation, and provide the DTO type and the type of the primary key.

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {
}
```

```
@Repository
public interface NoteRepository extends JpaRepository<Note, Long> {
}
```



JPA

JAVA PERSISTENCE API

We have a lot of methods available by default but we will also need custom queries. For example, we might want to get Customers by last name or by company.


This is easy, we just add the method declaration to our interface. No implementation needed. The names need only match the property names in our DTO:

```
public class Customer {
    private long id;
    private String lastName;
    private String company;

    //getters and setters omitted for brevity
}

@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    List<Customer> findByLastName(String lastName);
    List<Customer> findByCompany(String company);
}
```



Model Annotations

INDEPENDENT PRACTICE

It's time to fly. Focus. Work hard. Ask for help when you need it.

Work in PAIRS to complete all of the goals below.

Goals:

- Take a guess at how find customers by lastName and company. For example show all the employees with a last name of Patel at SpaceX.
- Recreate the Customer interface and add this new method.

Hint: JPA seems semi-magical. Follow the same pattern as before, but include both last name and company.

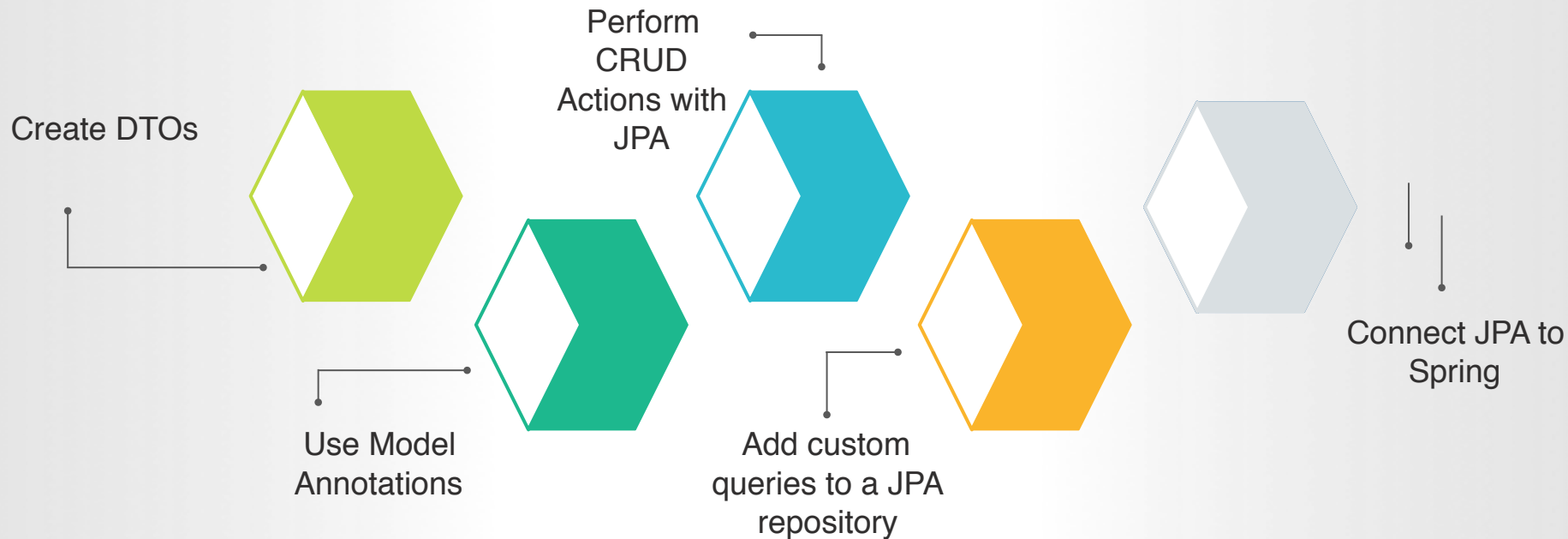


5 minutes!

Objectives & Key Outcomes

THE TAKEAWAYS FROM THIS CLASS

By the end of class today, you will be able to:

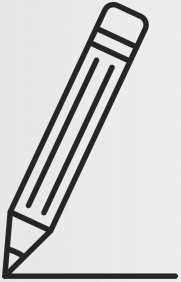


lunch.

Lab Time

TDD

TEST DRIVEN DEVELOPMENT



Notebooks Ready? It's time for a brief lecture.



TDD

TEST DRIVEN DEVELOPMENT

Before we begin our lab time, let's open the katas together.

Notice the test directory.

Test driven development is a development philosophy which entails creating tests before a single line of code is written.

Tests are written to fail and then code is written to make the tests pass.

We will discuss TDD more in depth in the next module, but in this module, we'll be providing opportunities for you to practice coding to pass tests.



LabTime

INDEPENDENT PRACTICE

It's time to fly. Focus. Work hard. Ask for help when you need it.

Work together but INDEPENDENTLY write your own code to complete all of the goals below.

Goals:

- Complete all the katas listed in the activity file.

If this is tough, great! You're getting practice.

If this is easy, great! Help your buddies.

We'll be circulating to provide individual help where it's needed.



**40
minutes!**



Stay Seated & Take 3 Deep Breaths.

RELAX.

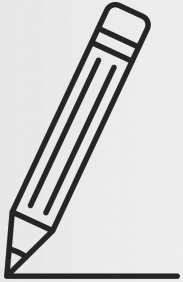
Now take a short walk. Clear your head. After a few minutes break, quickly review your notes.
We'll start back shortly.



JPA meets Spring

Spring Meets JPA

A LOVE STORY



Notebooks Ready? It's time for a brief lecture.



Spring Meets JPA

A LOVE STORY

Recall that our entire purpose as back-end developers is to safely and reliably get data from the database to the client and vice versa.

We've spent ample time studying using Spring Boot to get data to and from the client.

Today we've learned to use JPA to get data to and from a database.

Now let's marry the two and build a true full-stack application.

Note: That we will later add a layer between the web layer (REST API) and data layer (DTO and associated Repository) to handle logic.



Model Annotations

WATCH & LEARN

Close your laptop. Eyes on my screen. Pay attention.

```
@RestController  
  
public class CustomerController {  
  
    @Autowired  
    private CustomerRepository customerRepo;  
  
    @RequestMapping(value="/customers", method = RequestMethod.GET)  
    public List<Customer> getAllCustomers() {  
        return customerRepo.findAll();  
    }  
  
    ...  
  
}
```

Model Annotations

INDEPENDENT PRACTICE

It's time to fly. Focus. Work hard. Ask for help when you need it.

Work in PAIRS to complete all of the goals below.

Goals:

- Using the below schema, create a Motorcycle DTO.
- Create a Motorcycle Repository.
- Create a basic REST API for creating, reading, updating, and deleting motorcycles.

```
CREATE TABLE IF NOT EXISTS motorcycle (  
    id int NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    vin varchar(20) NOT NULL ,  
    make varchar(20) NOT NULL ,  
    model varchar(20) NOT NULL ,  
    year varchar(4) NOT NULL ,  
    color varchar(20) NOT NULL  
);
```

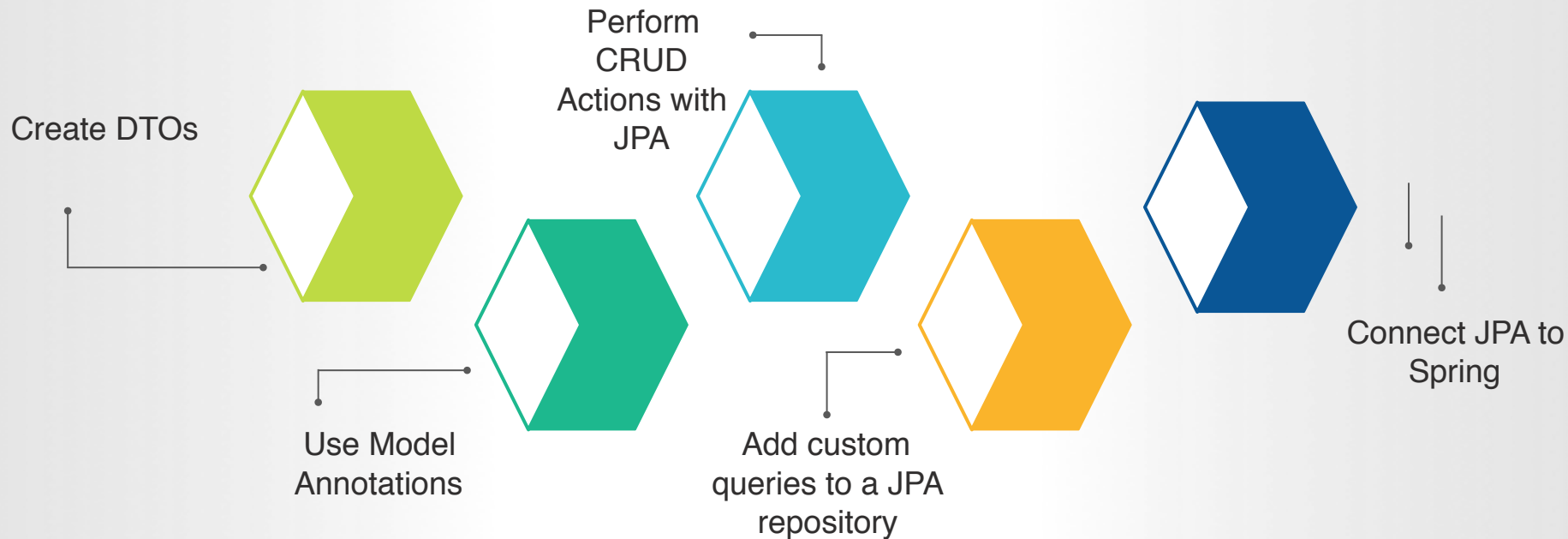


25
minutes!

Objectives & Key Outcomes


THE TAKEAWAYS FROM THIS CLASS

By the end of class today, you will be able to:



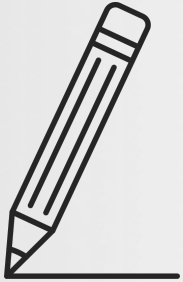
Sneak Peek

Note that next class will start with the sneak peek topic. You are NOT expected to master this today.



Service Layer

THE BRAINS OF THE OPERATION



Notebooks Ready? It's time for a brief lecture.



Service Layer

THE BRAINS OF THE OPERATION

The **service layer** acts as an intermediary between the **data layer** (JPA) and the **web layer** (REST Controller).

Where the web layer handles client requests, and the data layer saves and retrieves information in our database, the service layer provides the **business logic** for our application. For example, an algorithm which decides which videos to recommend for you on YouTube would live in the service layer.



Service Layer

WATCH & LEARN

Close your laptop. Eyes on my screen. Pay attention.

```
@Component
public class ServiceLayer {

    @Autowired
    private CustomerRepository customerRepo
    @Autowired
    private NoteRepository noteRepo;

    public Customer addCustomer(Customer customer) {
        customerRepo.save(customer);
        return customer;
    }

    public void updateCustomer(Customer customer, int id) {
        if (customer.getId() != id) {
            throw new IllegalArgumentException("Customer id must match id provided");
        }

        customerRepo.save(customer);
    }
}
```

Wrap Up

Module 4 Lesson 2

HOMEWORK

You don't have to submit your nightly homework, but you are expected to complete it.

For homework, read the below documentation on JPA and Annotations.

<https://docs.jboss.org/hibernate/stable/annotations/reference/en/html/entity.html>

Really read it. Take notes. If you come across something you're unfamiliar with, go down the rabbit hole and try to figure it out.



Daily Assessment Today

You may leave after a staff member approves your assessment.



Daily Assessment

Work INDIVIDUALLY to complete all of the goals below.

Goals:

- Create a new project with start.spring.io
- Create a DTO for the below schema
- Create a JPA repository for the schema
- Define an additional query method to find all books by author.

```
CREATE TABLE IF NOT EXISTS book (  
    id INTEGER AUTO_INCREMENT NOT NULL PRIMARY KEY,  
    title VARCHAR(80) NOT NULL,  
    author VARCHAR(80) NOT NULL,  
    checked_out BOOLEAN  
);
```