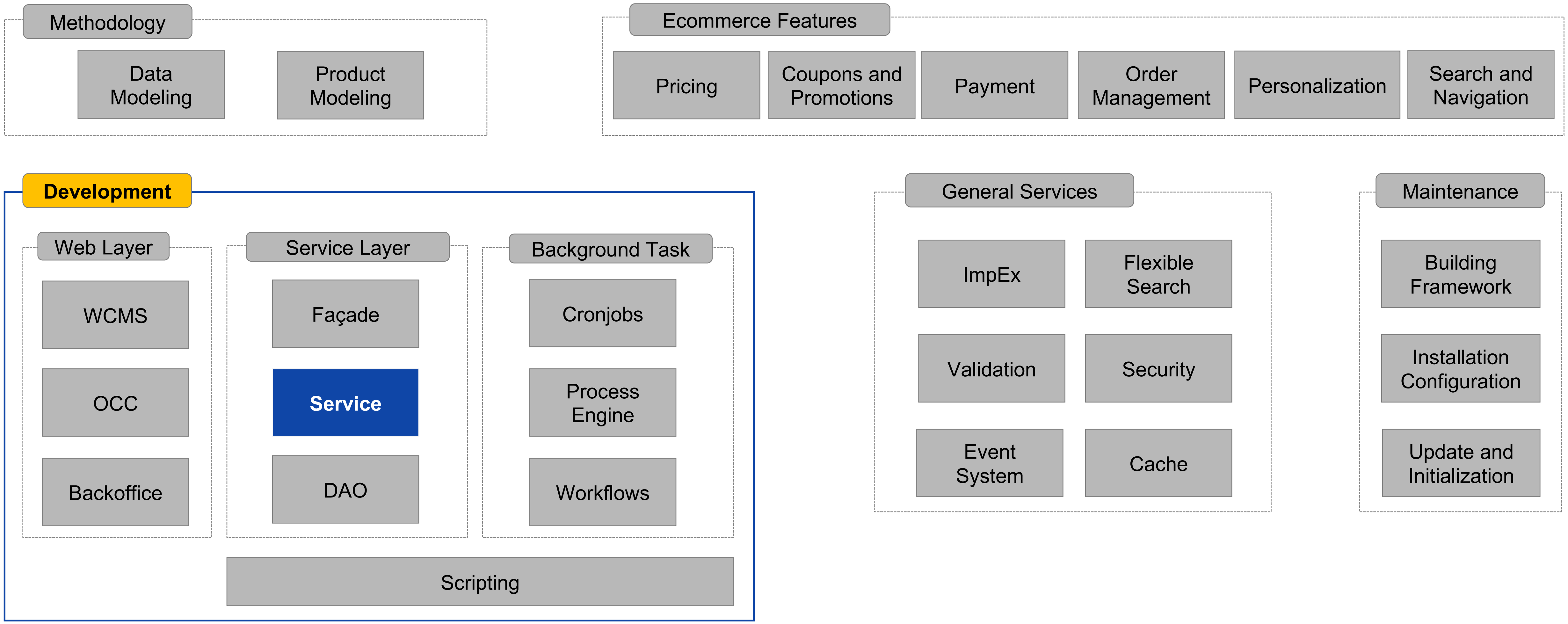
SAP Customer Experience

Services

What we will cover in this topic



We will learn about:

- ServiceLayer
- Commerce Services
- Models
- Platform Testing Environment
- Transactions
- ServiceLayer Direct

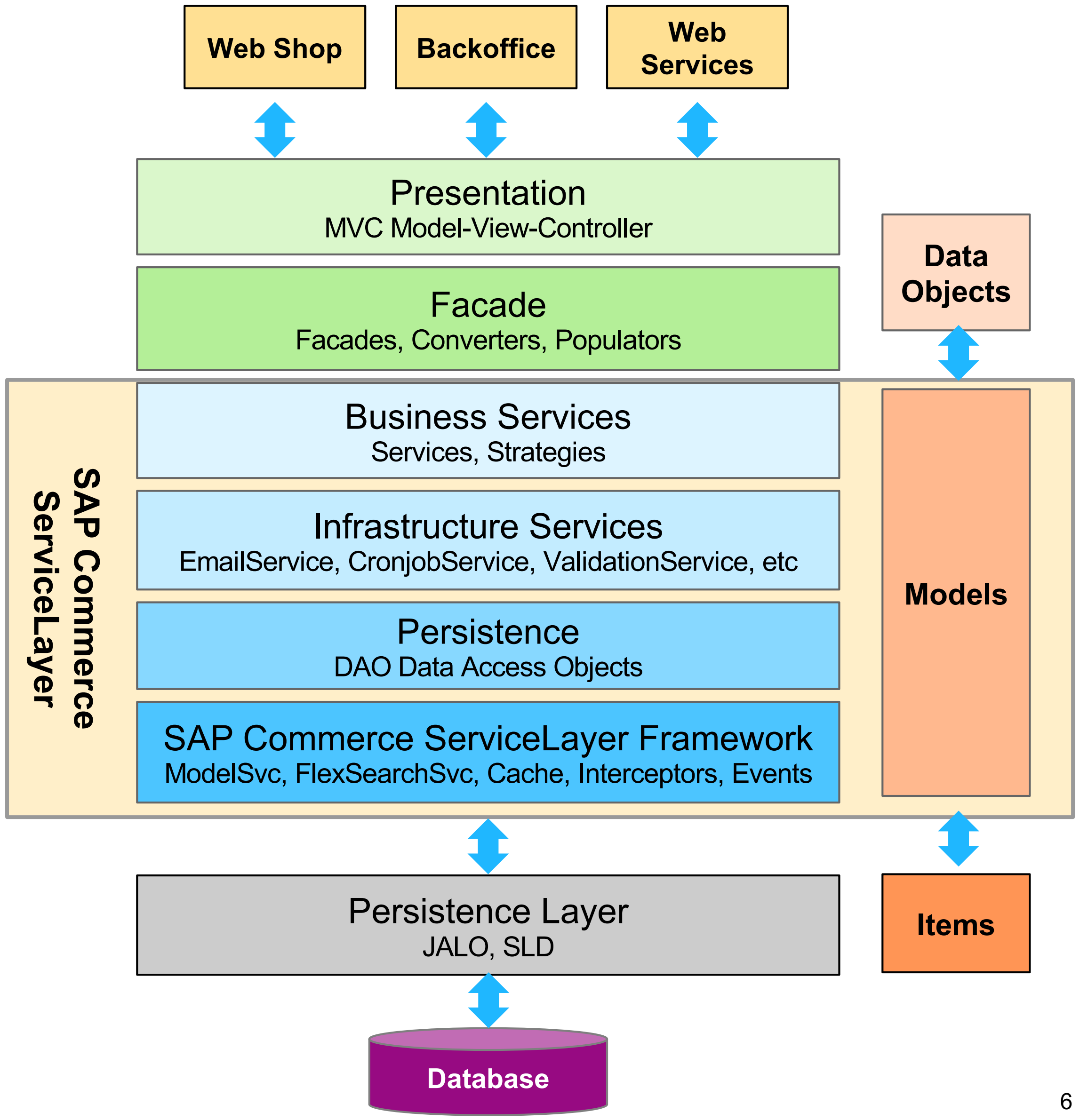
The Context

The SAP Commerce Cloud ServiceLayer provides a **Java API** for service development. It provides a number of standard services OOTB, which you can use or replace by either extending existing services or implementing your own.

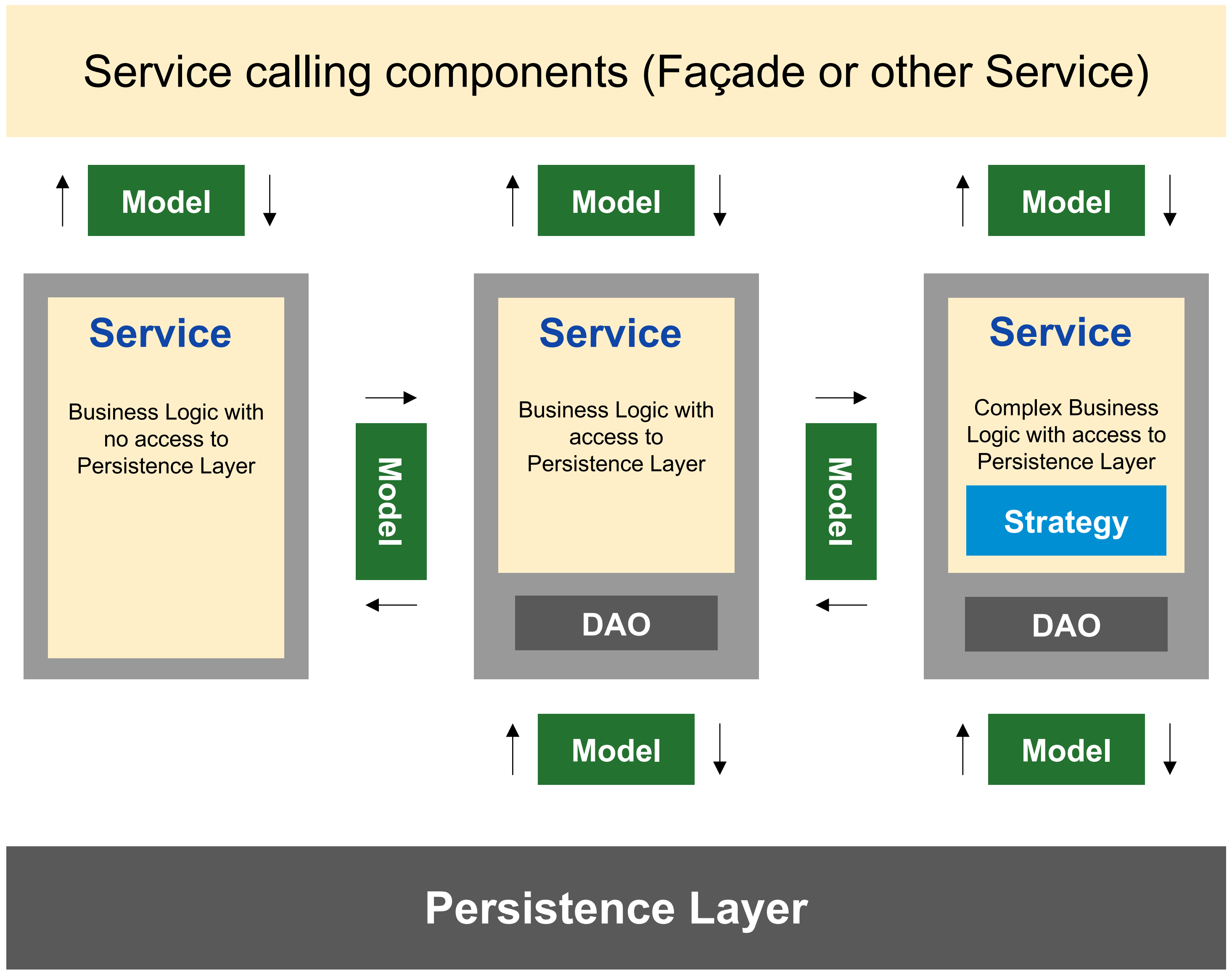
Architecture of the ServiceLayer

Overview of the SAP Commerce Cloud ServiceLayer

- The SAP Commerce architectural layer to implement *your* business logic
- Provides a number of services, each with well-defined responsibilities
- Service-oriented architecture based on the Spring framework
- Provides hooks into model life-cycle events to perform custom logic
- Provides a framework for publishing and receiving events


Web Shop
Backoffice
Web Services
Presentation
MVC Model-View-Controller
Facade
Facades, Converters, Populators
Data Objects
SAP Commerce ServiceLayer
Business Services
Services, Strategies
Infrastructure Services
EmailService, CronjobService, ValidationService, etc
Persistence
DAO Data Access Objects
SAP Commerce ServiceLayer Framework
ModelSvc, FlexSearchSvc, Cache, Interceptors, Events
Models
Persistence Layer
JALO, SLD
Items
Database

ServiceLayer – Structure and Data Objects


Service calling components (Façade or other Service)
Model
Model
Model
Service
Business Logic with no access to Persistence Layer
Model
Service
Business Logic with access to Persistence Layer
DAO
Model
Service
Complex Business Logic with access to Persistence Layer
Strategy
DAO
Model
Model
Persistence Layer

Using Services

- To implement your own business logic, you can:
  - Use existing services as is
  - Create your own services
  - Replace/extend/override existing services

- Each service in SAP Commerce is defined as a Spring bean and has a Spring alias
- To override an existing service :
  - Implement your service with the same interface
  - Register your service using the same alias in the Spring context

```
<alias alias="cartService"
       name="myCustomCartService" />

<bean id="myCustomCartService"
      class="my.project.MyCustomCartService" />
```

Configuring Services

- Services frequently need to call on other services or components
- Instead of fully configuring a new service from scratch, use the **parent** argument to inherit Spring configurations from the service bean it is extending

```
<bean id="myCustomCartService"
      class=" my.project.MyCustomCartService”
      parent=”defaultCartService" />
```

- You may override any property inherited from the parent, or use it as-is
- Special syntax exists for overriding or extending parent bean list values
- See **Reusing configuration from other beans** in the Spring Essentials for SAP Commerce
  - found in your handouts folder, under *Optional Reading.*

Commerce Services

commerceservices extension

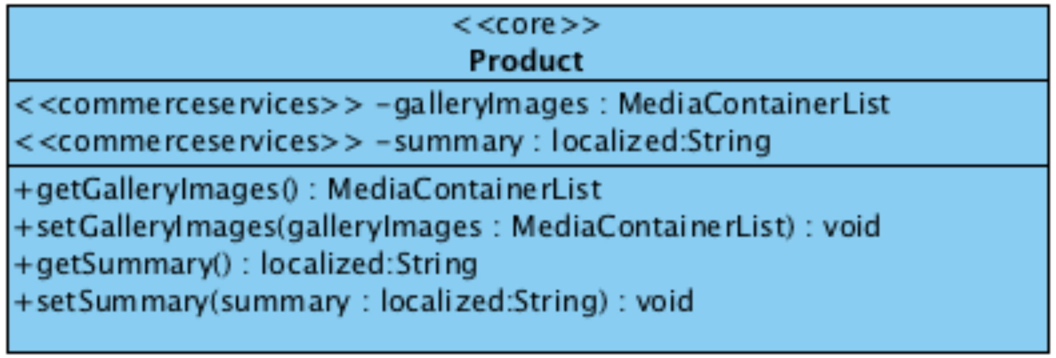
- Orchestrates platform and other extensions' services to provide complete **B2C use cases**
  - Example: The **commerceservices** extension provides the **CustomerAccountService**, which handles typical customer account management capabilities using the **userService**, **passwordEncoderService**, **baseStoreService**, and additional services from other extensions.

- Creates or extends more generic functionality from other extensions to **add more B2C features**
  - Example: The **commerceservices** extension extends the functionality of the **CartService** by creating the **CommerceCartService**, which adds promotions calculation, stock checks, and other Cart and Payment strategies to the base functionality.

For details, see: commerceservices Extension on //help.sap.com

Data Model: Product

The **commerceservices** extension also extends the platform data model by injecting new attributes into existing Types, e.g. Customer, Order or the **Product** types (and more):



```
<<commerceservices>> -galleryImages : MediaContainerList
<<commerceservices>> -summary : localized:String
+getGalleryImages() : MediaContainerList
+setGalleryImages(galleryImages : MediaContainerList) : void
+getSummary() : localized:String
+setSummary(summary : localized:String) : void
```

- galleryImages
  - storing multiple images each resized to a number of standard formats expected by the storefront
- summary
  - more concise product description (e.g. in storefront search)

Models

Overview of Models (I)

- Data objects the ServiceLayer is based on
- Each Item Type has a corresponding model class
- POJO-like objects, `Serializable` by default
- Providing attributes with getter and setter methods
- Generated during build

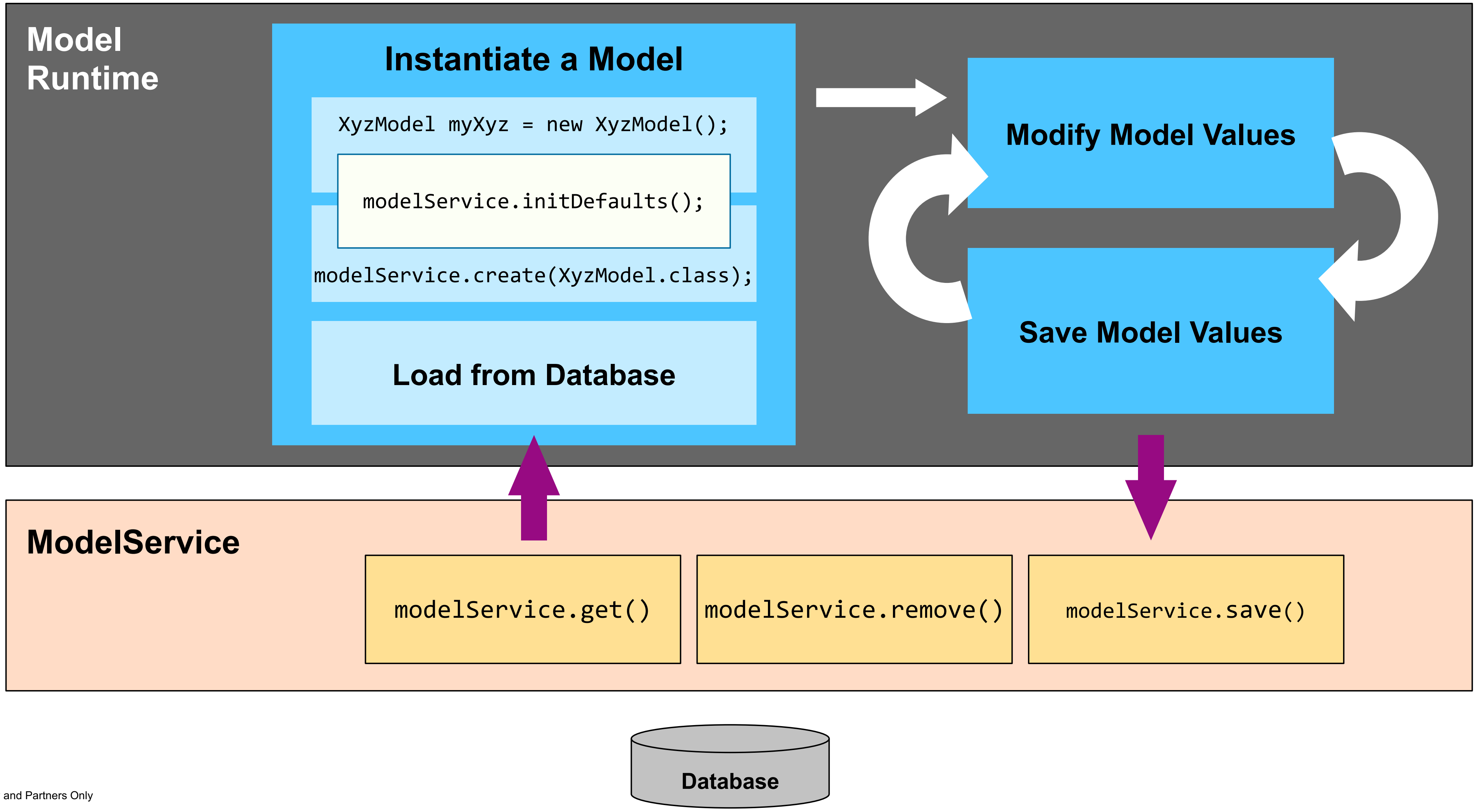
  ```
  ${HYBRIS_BIN_PATH}/platform/bootstrap/gensrc
  ```

Never manually edit SAP Commerce model classes!

Overview of Models (II)

- Models represent a certain “snapshot” of data from the database
  - No attachment to database: **representation is not live**
  - After modifying a model, you must explicitly save it back
- You may influence the extent to which lazy loading occurs on model/instance attributes (defaults to extremely lazy loading)
  - `servicelayer.prefetch` in `platform/resources/advanced.properties`

Lifecycle of a Model



Using Models

The **ModelService** bean deals with all aspects of a model's life-cycle

- Creating models
- Loading models by PK, from Items or via Flexible Search Query
- Updating / saving models
- Deleting models

Creation and processing of models

- Constructor:

Models created this way are neither filled with default values nor attached to the Model context

```
        ProductModel product = new ProductModel();
        modelService.initDefaults(product);
        modelService.save(product);  // The Model is saved to the database and automatically
attached
```

- Factory Method:

Models created this way are filled with default values and already attached to the Model context

```
        ProductModel product = modelService.create(ProductModel.class);
        modelService.save(product);
```

- Loading by PK

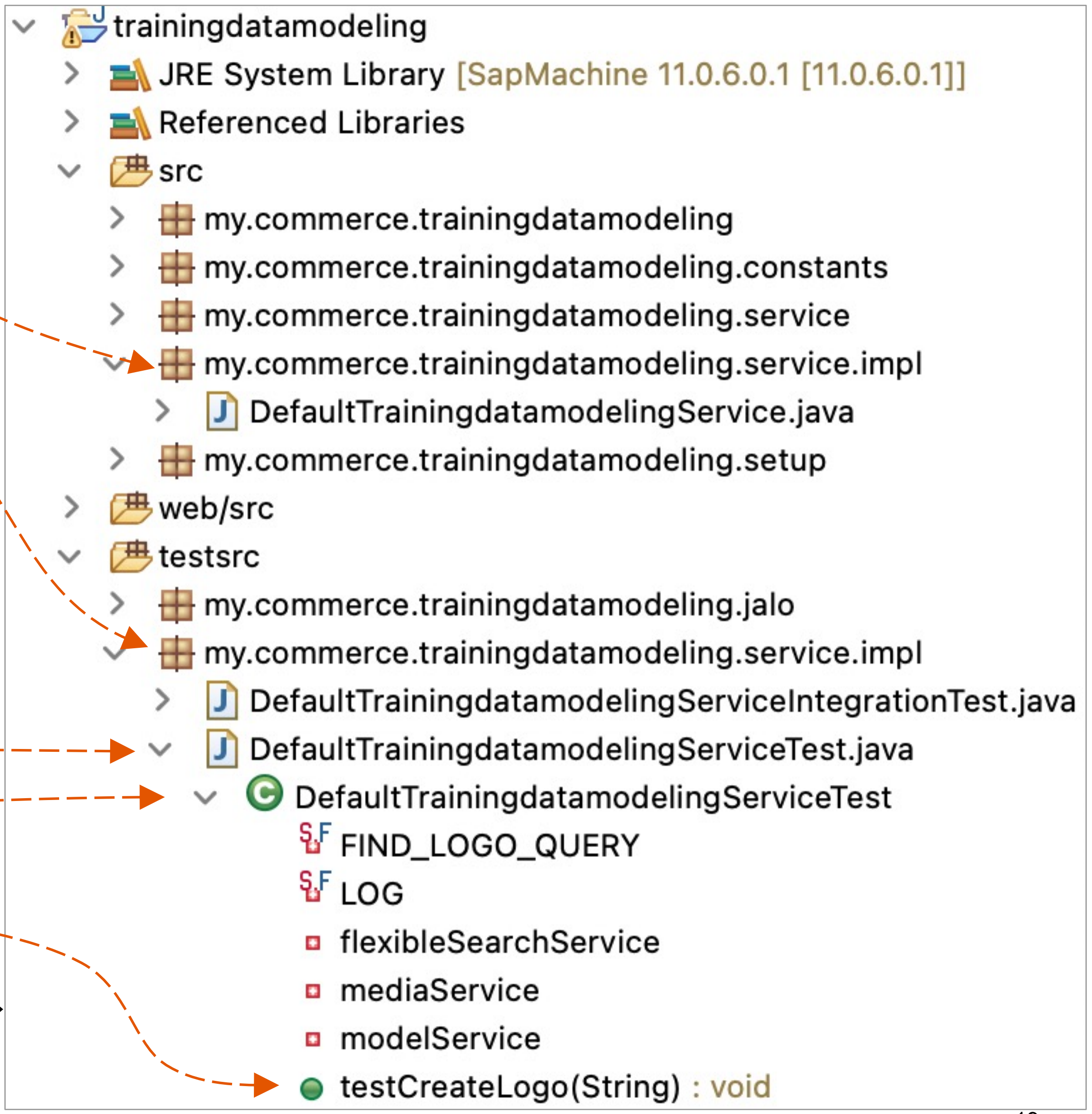
Models loaded this way are automatically attached to the Model context

```
        ProductModel product = modelService.get(pk);
```

Platform Testing Environment

JUnit Testing in SAP Commerce Cloud

- Standard practice: create corresponding test classes in parallel package inside **`testsrc`**
- Mark individual **methods** with **`@Test`** annotation (**`org.junit.Test`**)
- To invoke tests:
  - From Eclipse, invoke <u>Run As → JUnit Test</u> **Right-click** on…
    - Test class' **.java** node
    - Test class' **class** node
    - Individual method (annotated with **`@Test`**)
  - From command-line: **`ant <test target>`** (details to follow…)


```
trainingdatamodeling
  JRE System Library [SapMachine 11.0.6.0.1 [11.0.6.0.1]]
  Referenced Libraries
  src
    my.commerce.trainingdatamodeling
    my.commerce.trainingdatamodeling.constants
    my.commerce.trainingdatamodeling.service
    my.commerce.trainingdatamodeling.service.impl
      DefaultTrainingdatamodelingService.java
    my.commerce.trainingdatamodeling.setup
  web/src
  testsrc
    my.commerce.trainingdatamodeling.jalo
    my.commerce.trainingdatamodeling.service.impl
      DefaultTrainingdatamodelingServiceIntegrationTest.java
      DefaultTrainingdatamodelingServiceTest.java
        DefaultTrainingdatamodelingServiceTest
          FIND_LOGO_QUERY
          LOG
          flexibleSearchService
          mediaService
          modelService
          testCreateLogo(String) : void
```

Enabling JUnit Integration Tests in SAP Commerce Cloud

- Integration tests are “*tests that require runtime access to SAP Commerce Cloud services*” (e.g., `ModelService`, `FlexibleSearchService`, `YourCustomService`)

- Let test class extend `ServiceLayerTest` or `ServiceLayerTransactionalTest`
  - Initializes Spring context (use `@Resource`)
  - Useful methods inherited:
    - `createCoreData()` – provides basic SAP Commerce items for the test (language, currency, unit, country, etc.)
    - `createDefaultCatalog()` – sets up th default catalog (staged and online)
  - `ServiceLayerTransactionalTest` conveniently rolls back data changes

```
public class MySampleTest
   extends ServicelayerTest
{
   @Resource
   private ModelService modelSvc;

   @Resource
   private FlexibleSearchService flexSearchSvc;

   @Test
   public void test()
   {
      String fsqQuery = "SELECT {PK} from {Customer} "
                      + "where {uid}=?custId";
      FlexibleSearchQuery fsq =
            new FlexibleSearchQuery(fsqQuery);
      fsq.addQueryParameter("custId", "testCustomer");

      SearchResult<ProductModel> searchResult =
            flexSearchSvc.search(fsq);

      ... //assertTrue(...)
    }
}
```

Executing JUnit Tests

- Typically, a developer would execute JUnit tests from within an IDE, however…
- Ant can execute tests selectively using targets and command-line modifiers (some examples – modifiers can be combined):
  - `ant alltests -Dtestclasses.extensions=basecommerce,core`
  - `ant alltests -Dtestclasses.packages=de.hybris.platform.payment.*`
  - `ant alltests -Dtestclasses.annotations=unittests,integrationtests`
    (details on upcoming slide)

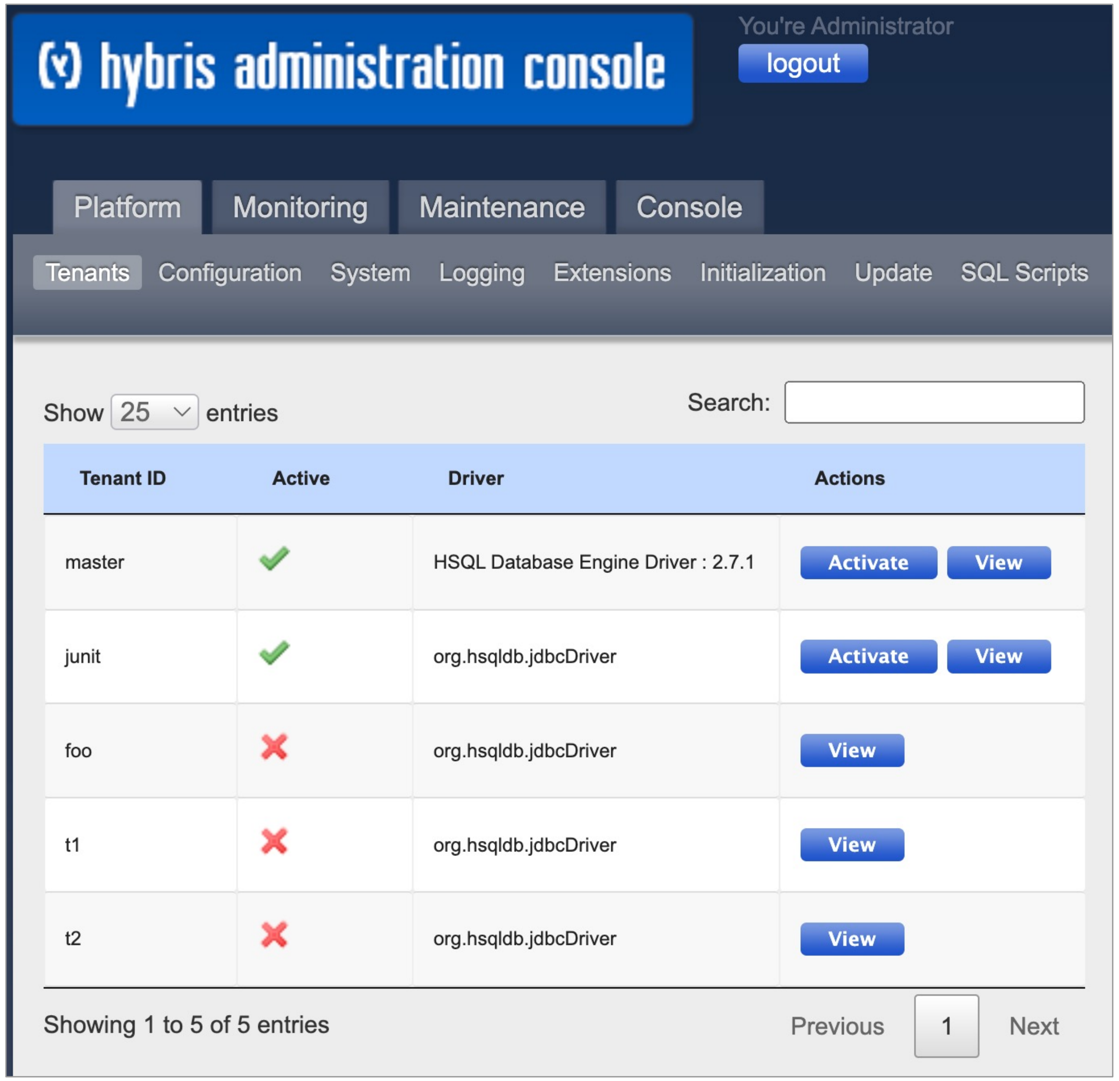
- SAP Commerce Testweb client
  - `http://localhost:9001/test`
  - Select tests by test type, extension, package, test class, etc., using web front-end

Additional `ant` targets for JUnit testing in SAP Commerce Cloud

- Class-level "filter" annotations
  (allow **ant** or Testweb front-end to specify which tests to run)
  - `ant alltests` – runs all test methods, ignoring all "filter" annotations
  - `ant integrationtests` – runs all test methods whose class is marked with `@IntegrationTest`
    Same as `ant alltests -Dtestclasses.annotations=integrationtests`
    NOTE: used in addition to class extending **`ServiceLayerTest`** or **`ServiceLayerTransactionalTest`**
  - `ant demotests` – runs all test methods whose class is marked with `@DemoTest`
    Same as `ant alltests -Dtestclasses.annotations=demotests`
  - `ant unittests` – runs all test methods whose class is marked with `@UnitTest`
    Same as `ant alltests -Dtestclasses.annotations=unittests`
  - `ant performancetests` – runs all test methods whose class is marked with `@PerformanceTest`
    Same as `ant alltests -Dtestclasses.annotations=performancetests`
  - `ant manualtests` – runs all test methods whose class is marked with `@ManualTest`
    Same as `ant alltests -Dtestclasses.annotations=manualtests`

The JUnit Tenant

- SAP Commerce Cloud includes a dedicated tenant for testing: the junit tenant
- ***Must*** be initialized before you can run tests:
  - You can invoke ***ant yunitinit*** in the platform directory
  - Or you can initialize it using HAC



Tenant ID	Active	Driver	Actions
master	✓	HSQL Database Engine Driver : 2.7.1	Activate View
junit	✓	org.hsqldb.jdbcDriver	Activate View
foo	✗	org.hsqldb.jdbcDriver	View
t1	✗	org.hsqldb.jdbcDriver	View
t2	✗	org.hsqldb.jdbcDriver	View

Showing 1 to 5 of 5 entries

Useful Reading

- Testing with JUnit

  https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/aa417173fe4a4ba5a473c93eb730a417/8c6e8668866910148fc390638f82bad2.html

- Unit Tests (Commerce 123)

  https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/d97b2ab46fde43a78640036ebf68e106/c26e305d91024b66acdc8572a00866b2.html

- Integration Tests (Commerce 123)

  https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/d97b2ab46fde43a78640036ebf68e106/91d6433b78ba47e59ddc1794917e269e.html

Transactions

Transactions

Transactions follow *ACID* paradigm (*atomic*, *consistent*, *isolated*, *durable*)

- The JTA UserTransaction interface is not provided
- SAP Commerce Cloud doesn't participate in global transactions of multiple systems
- Every Thread is only associated with 1 Transaction

Typically, 2 approaches to choose from:

Programmatic via API

- SAP Commerce cloud exposes underlying DB transaction system via simple API: Use begin, commit, and rollback statements in your code
  Note: Further encapsulation of the TA object using `TransactionBody` *is supported*

Declarative via Spring

- The Spring `PlatformTransactionManager` is implemented
- Access Spring TA Framework simply using `@Transactional`

More on ACID: Transactions – SAP Commerce Cloud video

Transactions: Transactions on //help.sap.com

```
// API sample
Transaction tx =
Transaction.current();
tx.begin();
boolean success = false;
try{
   // do business logic
   doSomeBusinessLogic();
   success = true;
}
finally{
   if (success) tx.commit();
   else tx.rollback();
}
```

Transaction annotation example

The preferred way over API approach to enable transactions in your code

- Better maintenance
- Reusability
- Testability
- Readability
- Less error-prone

Note:
Changing the isolation level will be ignored!

```
import
org.springframework.transaction.annotation.Transactional;


public class MySvc {
   @Transactional(~~isolation=Isolation.SERIALIZABLE~~)
   public void doTransactionalDbOperation() {
       modelService.save(productA);
       modelService.remove(productB);
   }
}
```

fixed to READ_COMMITED

ServiceLayer Direct

Capabilities

- SLD allows users to bypass the Jalo layer when writing/reading data to/from the database
- SLD can be enabled selectively when the use case allows it
- Fully transparent with the Model Layer
- Saving models uses transactions, optimistic locking and JDBC batch by default
- Less DB access operations - More efficient cache use

Note:
SLD is still disabled by default because of possible existing Jalo dependencies.
Careful testing is recommended

Enabling

- Globally, by setting `persistence.legacy.mode` property to false (default = true)

- Enable just for the current session context, even if disabled globally

```
PersistenceUtils.doWithSLDPersistence(() -> {
final TitleModel title = modelService.create(TitleModel.class);
title.setCode("foo");
modelService.save(title);
return title; });
```

- In Distributed ImpEx
  - by configuring `ImportConfig` object, applies to all imports

```
final ImportConfig config = new ImportConfig();

config.setDistributedImpexEnabled(true);  //enables distributed impex

config.setSldForData(true);  //enables direct persistence mode
```

  - Enable just for a selected batch or header, by setting modifier `sld.enabled=true`

```
INSERT_UPDATE Title[sld.enabled=true];code[unique=true]
;foo__sld_forced_by_header
;bar__sld_forced_by_header
```

Useful Reading

- An example of creating a service:

https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/aa417173fe4a4ba5a473c93eb730a417/8bcbf36d86691014b965bdd6abaefe5a.html

(Will help you with exercise Services)

- More on SLD

https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/aa417173fe4a4ba5a473c93eb730a417/ccf4dd14636b4f7eac2416846ffd5a70.html#enabling-servicelayer-direct-for-all-imports

https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD/aa417173fe4a4ba5a473c93eb730a417/d4dc444c9cbb46f78c78ff3d8a26539b.html

Key Points

1. The SAP Commerce Cloud Service layer is where you implement ***your* business logic**
2. The Service Layer implementation is based on the Spring framework
3. You can use, extend or replace existing services, or create your own services
4. **Models are POJO-like objects** and generated during ant build in `${HYBRIS_BIN_PATH}/platform/bootstrap/gensrc`
5. Use **`ModelService` for CRUD** operations on models
6. JUnit is supported for testing in SAP Commerce Cloud
7. Transactions can be managed via dedicated APIs or Spring.

Testing is important, but for a 4-day class, please focus on getting hands-on with SAP Commerce

Don't waste too much time fixing your test environment if you're getting errors.

Interceptors are very important concepts for managing models. More details are covered in a live session "SAP Commerce Cloud – Additional Technical Essentials"

Services Exercise

SAP Customer Experience

Thank you.