Table of Contents

Implementing the User Interface — Task-based

> To accomplish a difficult task, one must first make it easy.
>
> — Marty Rubin

The essence of Domain Driven Design(DDD) is a lot about capturing the business process and user intent a lot more closely. In the previous chapter, we designed a set of APIs without paying a lot of attention to how those APIs would get consumed by its eventual users. In this chapter, we will design the GUI for the LC application using the JavaFX[1] framework. As part of that, we will examine how this approach of designing APIs in isolation can cause an impedance mismatch between the producers and the consumers. We will examine the consequences of this *impedance mismatch* and how task-based UIs can help cope with this mismatch a lot better.

In this chapter, we will implement the UI for LC Application and wire up the integration to the backend APIs. The list of topics to be covered is as follows:

- API styles
- Implementing the UI

At the end of the chapter, you will learn how to employ DDD principles to help you build robust user experiences that are simple and intuitive. You will also learn why it may be prudent to design your backend interfaces (APIs) from the perspective of the consumer.

Technical requirements

To follow the examples in this chapter, you will need access to:

- JDK 1.8+ (We have used Java 16 to compile sample sources)
- JavaFX SDK 16 and SceneBuilder
- Spring Boot 2.4.x
- mvvmFX 1.8 (https://sialcasa.github.io/mvvmFX/)
- JUnit 5.7.x (Included with spring boot)
- TestFX (for UI testing)
- OpenJFX Monocle (for headless UI testing)
- Project Lombok (To reduce verbosity)
- Maven 3.x

Before we dive deep into building the GUI solution, let's do a quick recap of where we left the APIs.

API Styles

If you recall from chapter 5, we created the following commands:

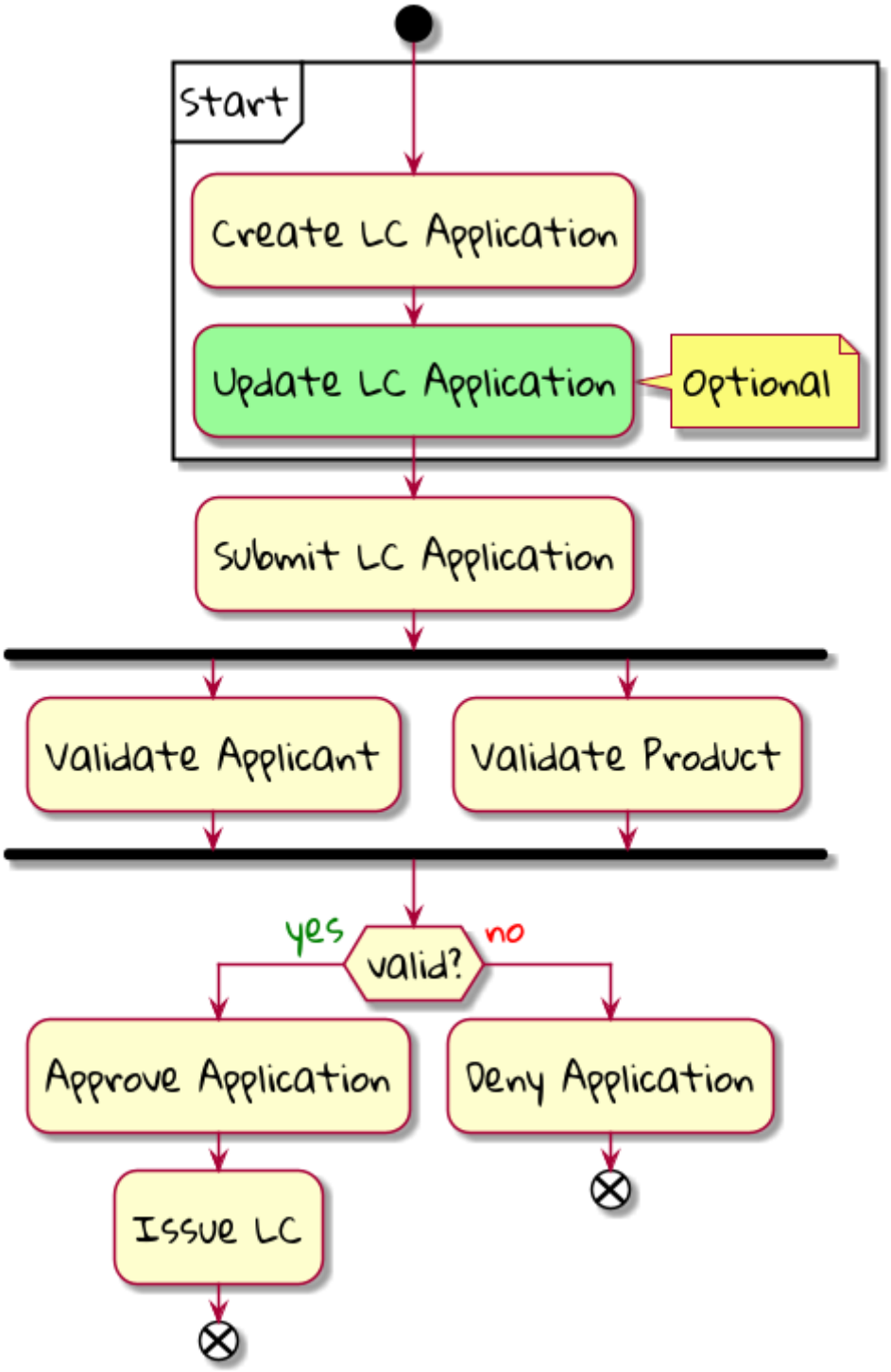

Figure 1. Commands from the event storming session

If you observe carefully, there seem to be commands at two levels of granularity. The "Create LC Application" and "Update LC application" are coarse grained, whereas the others are a lot more focused in terms of their intent. One possible decomposition of the coarse grained commands can be as depicted here:

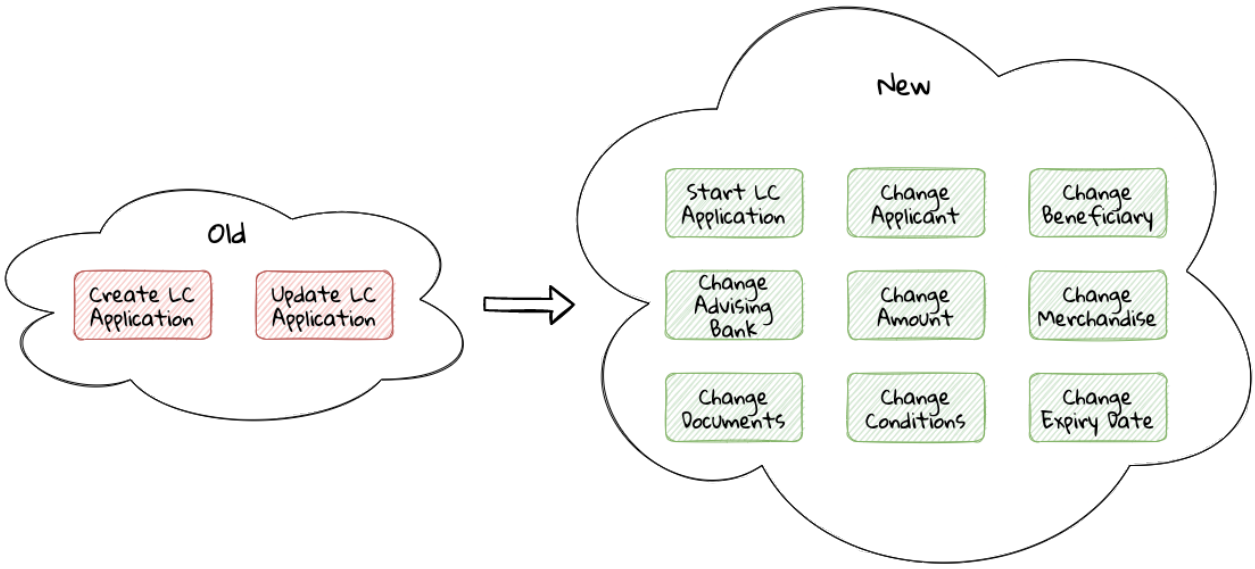

Figure 2. Decomposed commands

In addition to just being more fine-grained than the commands in the previous iteration, the revised commands seem to better capture the user's intent. This may feel like a minor change in semantics, but can have a huge impact on the way our solution is used by its ultimate end-users. The question then is whether we should *always* prefer fine-grained APIs over coarse grained ones. The answer can be a lot more nuanced. When designing APIs and experiences, we see two main styles being employed:

- CRUD-based
- Task-based

Let's look at each of these in a bit more detail:

CRUD-based APIs

CRUD is an acronym used to refer to the four basic operations that can be performed on database applications: Create, Read, Update, and Delete. Many programming languages and protocols have their own equivalent of CRUD, often with slight variations in naming and intent. For example, SQL — a popular language for interacting with databases — calls the four functions Insert, Select, Update, and Delete. Similarly, the HTTP protocol has `POST`, `GET`, `PUT` and `DELETE` as verbs to represent these CRUD operations. This approach has got extended to our design of APIs as well. This has resulted in the proliferation of both CRUD-based APIs and user experiences. Take a look at the `CreateLCApplicationCommand` from Chapter 5:

```
import lombok.Data;

@Data
public class CreateLCApplicationCommand {

    private LCApplicationId id;

    private ClientId clientId;
    private Party applicant;
    private Party beneficiary;
    private AdvisingBank advisingBank;
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;
}
```

Along similar lines, it would not be uncommon to create a corresponding `UpdateLCApplicationCommand` as depicted here:

```
import lombok.Data;

@Data
public class UpdateLCApplicationCommand {

    @TargetAggregateIdentifier
    private LCApplicationId id;

    private ClientId clientId;
    private Party applicant;
    private Party beneficiary;
    private AdvisingBank advisingBank;
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;
}
```

While this is very common and also very easy to grasp, it is not without problems. Here are some questions that taking this approach raises:

1. Are we allowed to change everything listed in the `update` command?
2. Assuming that everything can change, do they all change at the same time?
3. How do we know what exactly changed? Should we be doing a diff?
4. What if all the attributes mentioned above are not included in the `update` command?
5. What if we need to add attributes in future?
6. Is the business intent of what the user wanted to accomplish captured?

In a simple system, the answer to these questions may not matter that much. However, as system complexity increases, will this approach remain resilient to change? We feel that it merits taking a look at another approach called task-based APIs to be able to answer these questions.

Task-based APIs

In a typical organization, individuals perform tasks relevant to their specialization. The bigger the organization, the higher the degree of specialization. This approach of segregating tasks according to one's specialization makes sense, because it mitigates the possibility of stepping on each others' shoes, especially when getting complex pieces of work done. For example, in the LC application process, there is a need to establish the value/legality of the product while also determining the credit worthiness of the applicant. It makes sense that each of these tasks are usually performed by individuals in unrelated departments. It also follows that these tasks can be performed independently from the other.

In terms of a business process, if we have a single `CreateLCApplicationCommand` that precedes these operations, individuals in both departments firstly have to wait for the entire application to be filled out before either can commence their work. Secondly, if either piece of information is updated through a single `UpdateLCApplicationCommand`, it is unclear what changed. This can result in a spurious notification being sent to at least one department because of this lack of clarity in the process.

Since most work happens in the form of specific tasks, it can work to our advantage if our processes and by extension, our APIs mirror these behaviors.

Keeping this in mind, let's re-examine our revised APIs for the LC application process:

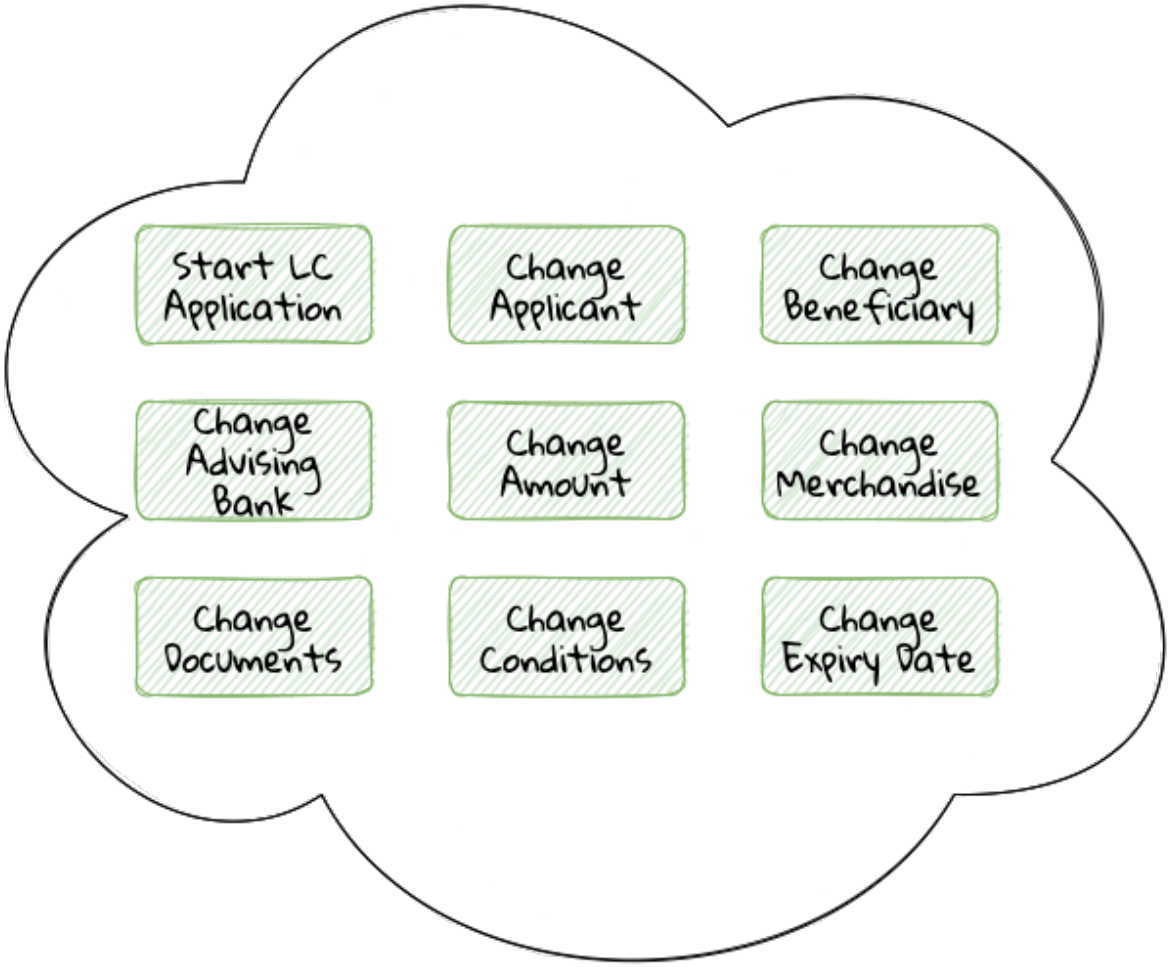

Figure 3. Revised commands

While it may have appeared previously that we have simply converted our coarse-grained APIs to become more fine-grained, this in reality is a better representation of the tasks that the user intended to perform. So, in essence, task-based APIs are the decomposition of work in a manner that aligns more closely to the users' intents. With our new APIs, product validation can commence as soon as `ChangeMerchandise` happens. Also, it is unambiguously clear what the user did and what needs to happen in reaction to the user's action. It then begs the question on whether we should employ task-based APIs all the time? Let's look at the implications in more detail.

Task-based or CRUD-based?

CRUD-based APIs seem to operate at the level of the aggregate. In our example, we have the LC aggregate. In the simplest case, this essentially translates to four operations aligned with each of the CRUD verbs. However, as we are seeing, even in our simplified version, the LC is becoming a fairly complex concept. Having to work with just four operations at the level of the LC is cognitively complex. With more requirements, this complexity will only continue to increase. For example, consider a situation where the business expresses a need to capture a lot more information about the `merchandise`, where today, this is simply captured in the form of free-form text. A more elaborate version of merchandise information is shown here:

```
public class Merchandise {
    private MerchandiseId id;
    private Set<Item> items;
    private Packaging packaging;
    private boolean hazardous;
}

class Item {
    private ProductId productId;
    private int quantity;
    // ...
}

class Packaging {
    // ...
}
```

In our current design, the implications of this change are far reaching for both the provider and the consumer(s). Let's look at some of the consequences in more detail:

Characteristic	CRUD-based	Task-based	Commentary
Usability	👎	👍	Task-based interfaces tend to provide more fine-grained controls that capture user intent a lot more explicitly, making them naturally more usable — especially in cases where the domain is complex.
Reusability	👎	👍	Task-based interfaces enable more complex features to be composed using simpler ones providing more flexibility to the consumers.
Scalability	👎	👍	Task-based interfaces have an advantage because they can provide the ability to independently scale specific features. However, if the fine-grained task-based interfaces are used almost all the time in unison, it may be required to re-examine whether the users' intents are accurately captured.
Security	👎	👍	For task-based interfaces, security is enhanced from the producer's perspective by enabling application of the *principle of least privilege*[2].
Complexity	👎	👍	Complexity of the system as a whole is proportional to the number of features that need to be implemented. Assuming accidental complexity is avoided in both cases, task-based interfaces allow spreading complexity more or less uniformly across multiple simpler interfaces.
Latency	👍	👎	Arguably, coarse-grained CRUD interfaces can enable consumers to achieve a lot more in less interactions, thereby providing low latency.

Characteristic	CRUD-based	Task-based	Commentary
Management Overhead	👍	👎	For the provider, fine-grained interfaces require a lot more work managing a larger number of interfaces.

As we can see, the decision between CRUD-based and task-based interfaces is nuanced. We are not suggesting that you should choose one over the other. Which style you use will depend on your specific requirements and context. In our experience, task-based interfaces treat user intents as first class citizens and perpetrate the spirit of DDD's ubiquitous language very elegantly. Our preference is to design interfaces as task-based where possible, because they result in more intuitive interfaces that better express the problem domain.

As systems evolve, and the support richer user experiences and multiple channels, CRUD-based seem to require additional translation layers to cater to user experience needs. The visual here depicts a typical layered architecture of a solution that supports multiple user experience channels:

[bff] | *ui-patterns/bff.png*

This set up is usually composed of:

1. Domain tier comprised of CRUD-based services that simply map closely to database entities.
2. Composite tier comprised of business capabilities that span more than one core service.
3. Backend-for-frontend (BFF[3]) tier comprised of channel-specific APIs.

Do note that the composite and BFF tiers exist primarily as a means to map backend capabilities to user intent. In an ideal world, where backend APIs reflect user intent closely, the need for translations should be minimal (if at all). Our experience suggests that such a setup causes business logic to get pushed closer to the user channels as opposed to being encapsulated within the confines of well-factored business services. In addition, these tiers cause inconsistent experiences across channels for the same functionality, given that modern teams are structured along tier boundaries.

We are not opposed to the use of layered architectures. We recognize that a layered architecture can bring modularity, separation of concerns and other related benefits. However, we are opposed to creating additional tiers merely as a means to compensate for poorly factored core domain APIs.

A well factored API tier can have a profound effect on how great user experiences are built. However, this is a chapter on implementing the user interface. Let's revert to creating the user interface for the LC application.

Bootstrapping the UI

We will be building the UI for the LC issuance application we created in Chapter 5: Implementing Domain Logic. For detailed instructions, refer to the section on *Bootstrapping the application*. In addition, we will need to add the following dependencies to the `dependencies` section of the Maven `pom.xml` file in the root directory of the project:

```
<dependencies>
    <!--...-->
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-controls</artifactId>
        <version>${javafx.version}</version>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-graphics</artifactId>
        <version>${javafx.version}</version>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-fxml</artifactId>
        <version>${javafx.version}</version>
    </dependency>
    <dependency>
        <groupId>de.saxsys</groupId>
        <artifactId>mvvmfx</artifactId>
        <version>${mvvmfx.version}</version>
    </dependency>
    <dependency>
        <groupId>de.saxsys</groupId>
        <artifactId>mvvmfx-spring-boot</artifactId>
        <version>${mvvmfx.version}</version>
    </dependency>
    <!--...-->
</dependencies>
```

To run UI tests, you will need to add the following dependencies:

```
<dependencies>
    <!--...-->
    <dependency>
        <groupId>org.testfx</groupId>
        <artifactId>testfx-junit5</artifactId>
        <scope>test</scope>
        <version>${testfx-junit5.version}</version>
    </dependency>
    <dependency>
        <groupId>org.testfx</groupId>
        <artifactId>openjfx-monocle</artifactId>
        <version>${openjfx-monocle.version}</version>
    </dependency>
    <dependency>
        <groupId>de.saxsys</groupId>
        <artifactId>mvvmfx-testing-utils</artifactId>
        <version>${mvvmfx.version}</version>
        <scope>test</scope>
    </dependency>
    <!--...-->
</dependencies>
```

To be able to run the application from the command line, you will need to add the `javafx-maven-plugin` to the `plugins` section of your `pom.xml`, per the following:

```
<plugin>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-maven-plugin</artifactId>
    <version>${javafx-maven-plugin.version}</version>
    <configuration>
        <mainClass>com.premonition.lc.ch06.App</mainClass>
    </configuration>
</plugin>
```

To run the application from the command line, use:

```
mvn javafx:run
```

If you are using a JDK greater that version 1.8, the JavaFX libraries may not be bundled with the JDK itself. When running the application from your IDE, you will likely need to add the following:

```
--module-path=<path-to-javafx-sdk>/lib/ \
   --add-modules=javafx.controls,javafx.graphics,javafx.fxml,javafx.media
```

We are making use of the mvvmFX framework to assemble the UI. To make this work with spring boot, the application launcher looks as depicted here:

```
@SpringBootApplication
public class App extends MvvmfxSpringApplication { ①

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void startMvvmfx(Stage stage) {
        stage.setTitle("LC Issuance");

        final Parent parent = FluentViewLoader
                .fxmlView(MainView.class)
                .load().getView();

        final Scene scene = new Scene(parent);
        stage.setScene(scene);
        stage.show();
    }
}
```

① Note that we are required to extend from the mvvmFX framework class `MvvmfxSpringApplication`.

Please refer to the ch06 directory of the accompanying source code repository for the complete example.

Implementing the UI

When working with user interfaces, it is fairly customary to use one of these presentation patterns:

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)

The MVC pattern has been around for the longest time. The idea of separating concerns among collaborating model, view and controller objects is a sound one. However, beyond the definition of these objects, actual implementations seem to vary wildly — with the controller becoming overly complex in a lot of cases. In contrast, MVP and MVVM, while being derivatives of MVC, seem to bring out better separation of concerns between the collaborating objects. MVVM, in particular when coupled with data binding constructs, make for code that is much more readable, maintainable and testable. In this book, we make use of MVVM because it enables test-driven development which is a strong personal preference for us. Let's look at a quick MVVM primer as implemented in the `mvvmfx` framework.

Model View View-Model (MVVM) primer

Modern UI frameworks started adopting a declarative style to express the view. MVVM was designed to remove all GUI code (code-behind) from the view by making use of binding expressions. This allowed for a cleaner separation of stylistic vs. programming concerns. A high level visual of how this pattern is implemented is shown here:

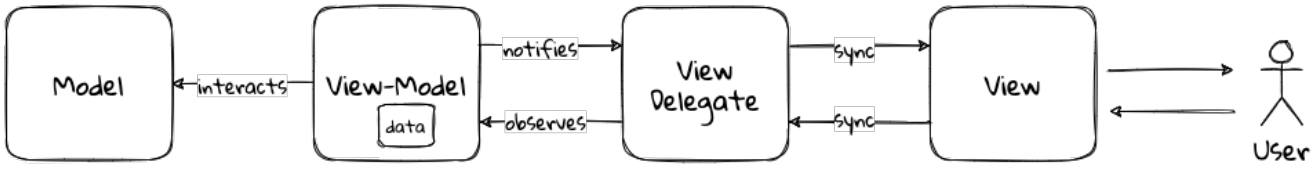

Figure 4. MVVM design pattern

The pattern comprises the following components:

- **Model**: responsible to house the business logic and managing the state of the application.
- **View**: responsible for presenting data to the user and notifying the view-model about user interactions through the view delegate.
- **View Delegate**: responsible for keeping the view and the view model in sync as changes are made by the user or on the view model. It is also responsible for transmitting actions performed on the view to the view model.
- **View-Model**: responsible for handling user interactions on behalf of the view. The view-model interacts with the view using the observer pattern (typically one-way or two-way data binding to make it more convenient). The view-model interacts with the model for updates and read operations.

Creating a new LC

Let's consider the example of creating a new LC. To start creation of a new LC, all we need is for the applicant to provide a friendly client reference. This is an easy to remember string of free text. A simple rendition of this UI is shown here:

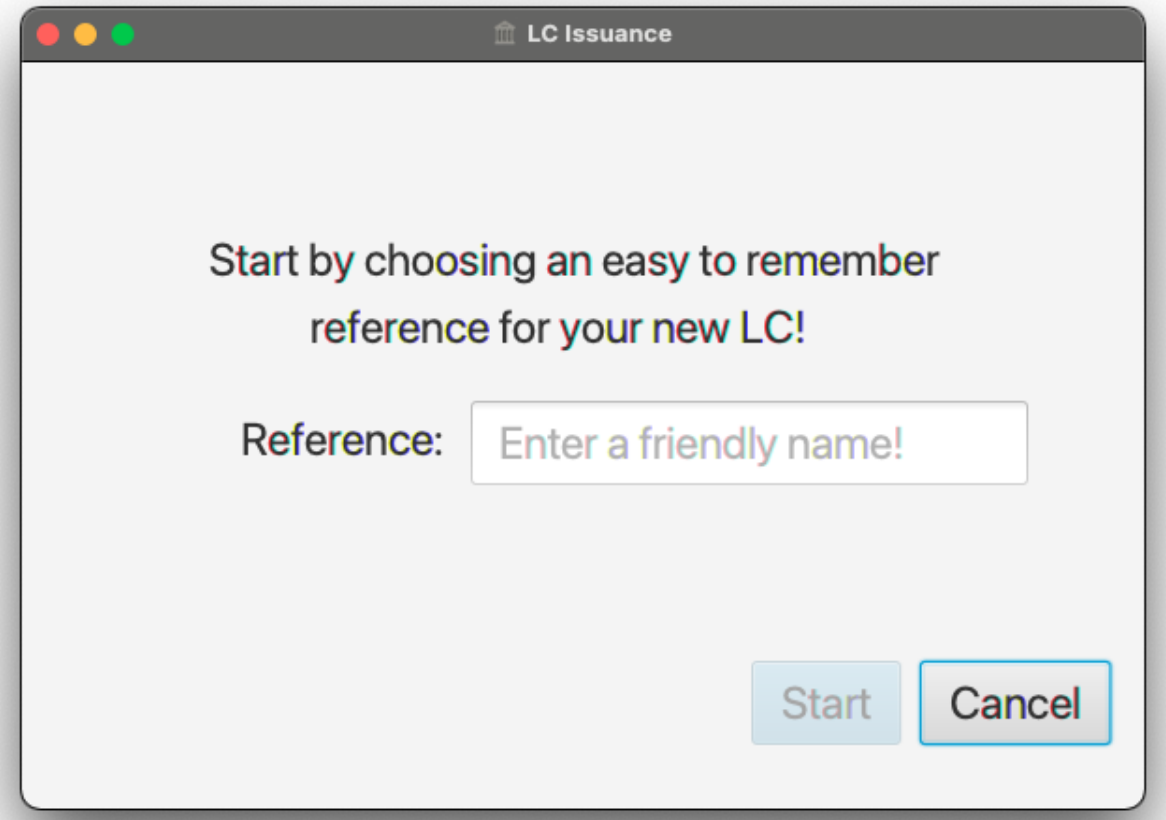

Figure 5. Start LC creation screen

Let's examine the implementation and purpose of each component in more detail.

Declarative view

When working with JavaFX, the view can be rendered using a declarative style in FXML format. Important excerpts from the `StartLCView.fxml` file to start creating a new LC are shown here:

```
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.TextField?>

<Pane id="start-lc"  xmlns="http://javafx.com/javafx/16"
                      xmlns:fx="http://javafx.com/fxml/1"
      fx:controller="com.premonition.lc.ch06.ui.views.StartLCView">      ①
    ...

    <TextField id="client-reference"
               fx:id="clientReference"/>                                  ②

    <Button id="start-button"
            fx:id="startButton"
            text="Start"
            onAction="#start"/>                                         ③
    ...
</Pane>
```

① The `StartLCView` class acts as the view delegate for the FXML view and is assigned using the `fx:controller` attribute of the root element (`javafx.scene.layout.Pane` in this case).

② In order to reference `client-reference` input field in the view delegate, we use the `fx:id` annotation — `clientReference` in this case.

③ Similarly, the `start-button` is referenced using `fx:id="startButton"` in the view delegate. Furthermore, the `start` method in the view delegate is assigned to handle the default action (the button press event for `javafx.scene.control.Button`).

View delegate

Next, let's look at the structure of the view delegate `com.premonition.lc.issuance.ui.views.StartLCView`:

```
import javafx.fxml.FXML;
//...
public class StartLCView {                      ①

    @FXML
    private TextField clientReference;          ②
    @FXML
    private Button startButton;                 ③

    public void start(ActionEvent event) {      ④
        // Handle button press logic here
    }

    // Other parts omitted for brevity...
}
```

① The view delegate class for the `StartLCView.fxml` view.

② The Java binding for the `clientReference` textbox in the view. The name of the member needs to match exactly with the value of the `fx:id` attribute in the view. Further, it needs to be annotated with the `@javafx.fxml.FXML` annotation. The use of the `@FXML` annotation is optional if the member in the view delegate is `public` and matches the name in the view.

③ Similarly, the `startButton` is bound to the corresponding button widget in the view.

④ The method for the action handler when the `startButton` is pressed.

View-Model

The view-model class `StartLCViewModel` for the `StartLCView` is shown here:

```
import javafx.beans.property.StringProperty;
import de.saxsys.mvvmfx.ViewModel;

public class StartLCViewModel implements ViewModel {          ①

    private final StringProperty clientReference;              ②

    public StartLCViewModel() {
        this.clientReference = new SimpleStringProperty();     ③
    }

    public StringProperty clientReferenceProperty() {          ④
        return clientReference;
    }

    public String getClientReference() {
        return clientReference.get();
    }

    public void setClientReference(String clientReference) {
        this.clientReference.set(clientReference);
    }

    // Other getters and setters omitted for brevity
}
```

① The view-model class for the `StartLCView`. Note that we are required to implement the `de.saxsys.mvvmfx.ViewModel` interface provided by the mvvmFX framework.

② We are initializing the `clientReference` property using the `SimpleStringProperty` provided by JavaFX. There are several other property classes to define more complex types. Please refer to the JavaFX documentation for more details.

③ The value of the `clientReference` in the view-model. We will look at how to associate this with value of the `clientReference` textbox in the view shortly. Note that we are using the `StringProperty` provided by `JavaFX`, which provides access to the underlying `String` value of the client reference.

④ `JavaFX` beans are required to create a special accessor for the property itself in addition to the standard getter and setter for the underlying value.

Binding the view to the view-model

Next, let's look at how to associate the view to the view-model:

```
import de.saxsys.mvvmfx.Initialize;
import de.saxsys.mvvmfx.FxmlView;
import de.saxsys.mvvmfx.InjectViewModel;
//...
public class StartLCView implements FxmlView<StartLCViewModel> {     ①

    @FXML
    private TextField clientReference;
    @FXML
    private Button startButton;

    @InjectViewModel
    private StartLCViewModel viewModel;                              ②

    @Initialize
    private void initialize() {                                      ③
        clientReference.textProperty()
            .bindBidirectional(viewModel.clientReferenceProperty()); ④
        startButton.disableProperty()
            .bind(viewModel.startDisabledProperty());                ⑤
    }

    // Other parts omitted for brevity...
}
```

① The mvvmFX framework requires that the view delegate implement the `FXMLView<? extends ViewModelType>`. In this case, the view-model type is `StartLCViewModel`. The mvvmFX framework supports other view types as well. Please refer to the framework documentation for more details.

② The framework provides a `@de.saxsys.mvvmfx.InjectViewModel` annotation to allow dependency injecting the view-model into the view delegate.

③ The framework will invoke all methods annotated with the `@de.saxsys.mvvmfx.Initialize` annotation during the initialization process. The annotation can be omitted if the method is named `initialize` and is declared `public`. Please refer to the framework documentation for more details.

④ We have now bound the text property of the `clientReference` textbox in the view delegate to the corresponding property in the view-model. Note that this is a **bidirectional** binding, which means that the value in the view and the view model are kept in sync if it changes on either side.

⑤ This is another variation of binding in action, where we are making use of a unidirectional binding. Here, we are binding the disabled property of the `start` button to the corresponding

property on the view-model. We will look at why we need to do this shortly.

Enforcing business validations in the UI

We have a business validation that the client reference for an LC needs to be at least 4 characters in length. This will be enforced on the back-end. However, to provide a richer user experience, we will also enforce this validation on the UI.

This may feel contrary to the notion of centralizing business validations on the back-end. While this may be a noble attempt at implementing the DRY (Don't Repeat Yourself) principle, in reality, it poses a lot of practical problems. Distributed systems expert — Udi Dahan has a very interesting take on why this may not be such a virtuous thing to pursue[4]. Ted Neward also talks about this in his blog titled *The Fallacies of Enterprise Computing*[5].

The advantage of using MVVM is that this logic is easily testable in a simple unit test of the view-model. Let's see this in action test-drive this now:

```
class StartLCViewModelTests {

    private StartLCViewModel viewModel;

    @BeforeEach
    void before() {
        int clientReferenceMinLength = 4;
        viewModel = new StartLCViewModel(clientReferenceMinLength);
    }

    @Test
    void shouldNotEnableStartByDefault() {
        assertThat(viewModel.getStartDisabled()).isTrue();
    }

    @Test
    void shouldNotEnableStartIfClientReferenceLesserThanMinimumLength() {
        viewModel.setClientReference("123");
        assertThat(viewModel.getStartDisabled()).isTrue();
    }

    @Test
    void shouldEnableStartIfClientReferenceEqualToMinimumLength() {
        viewModel.setClientReference("1234");
        assertThat(viewModel.getStartDisabled()).isFalse();
    }

    @Test
    void shouldEnableStartIfClientReferenceGreaterThanMinimumLength() {
        viewModel.setClientReference("12345");
        assertThat(viewModel.getStartDisabled()).isFalse();
    }
}
```

Now, let's look at the implementation for this functionality in the view-model:

```
public class StartLCViewModel implements ViewModel {

    //...
    private final StringProperty clientReference;
    private final BooleanProperty startDisabled;                    ①

    public StartLCViewModel(int clientReferenceMinLength) {         ②
        this.clientReference = new SimpleStringProperty();
        this.startDisabled = new SimpleBooleanProperty();
        this.startDisabled
            .bind(this.clientReference.length()
                    .lessThan(clientReferenceMinLength));           ③
    }

    //...
}

public class StartLCView implements FxmlView<StartLCViewModel> {

    //...
    @Initialize
    public void initialize() {
        startButton.disableProperty()
            .bind(viewModel.startDisabledProperty());               ④
        clientReference.textProperty()
            .bindBidirectional(viewModel.clientReferenceProperty());
    }
    //...
}
```

① We declare a `startDisabled` property in the view-model to manage when the start button should be disabled.

② The minimum length for a valid client reference is injected into the view-model. It is conceivable that this value will be provided as part of external configuration, or possibly from the back-end.

③ We create a binding expression to match the business requirement.

④ We bind the view-model property to the disabled property of the start button in the view delegate.

Let's also look at how to write an end-to-end, headless UI test as shown here:

```
@UITest
public class StartLCViewTests {                                    ①

    @Autowired
    private ApplicationContext context;

    @Init
    public void init() {
        MvvmFX.setCustomDependencyInjector(context::getBean);      ②
    }

    @Start
    public void start(Stage stage) {                               ③
        final Parent parent = FluentViewLoader
                .fxmlView(StartLCView.class)
                .load().getView();
        stage.setScene(new Scene(parent));
        stage.show();
    }

    @Test
    void blankClientReference(FxRobot robot) {
        robot.lookup("#client-reference")                          ④
            .queryAs(TextField.class)
            .setText("");

        verifyThat("#start-button", NodeMatchers.isDisabled());    ⑤
    }

    @Test
    void validClientReference(FxRobot robot) {
        robot.lookup("#client-reference")
            .queryAs(TextField.class)
            .setText("Test");

        verifyThat("#start-button", NodeMatchers.isEnabled());     ⑤
    }
}
```

① We have written a convenience `@UITest` extension to combine spring framework and TestFX testing. Please refer to the accompanying source code with the book for more details.

② We set up the spring context to act as the dependency injection provider for the mvvmFX framework and its injection annotations (for example, `@InjectViewModel`) to work.

③ We are using the `@Start` annotation provided by the TestFX framework to launch the UI.

④ The TestFX framework injects an instance of the `FxRobot` UI helper, which we can use to access UI elements.

⑤ We are using the The TestFX framework provided convenience matchers for test assertions.

Now, when we run the application, we can see that the start button is enabled when a valid client reference is entered:

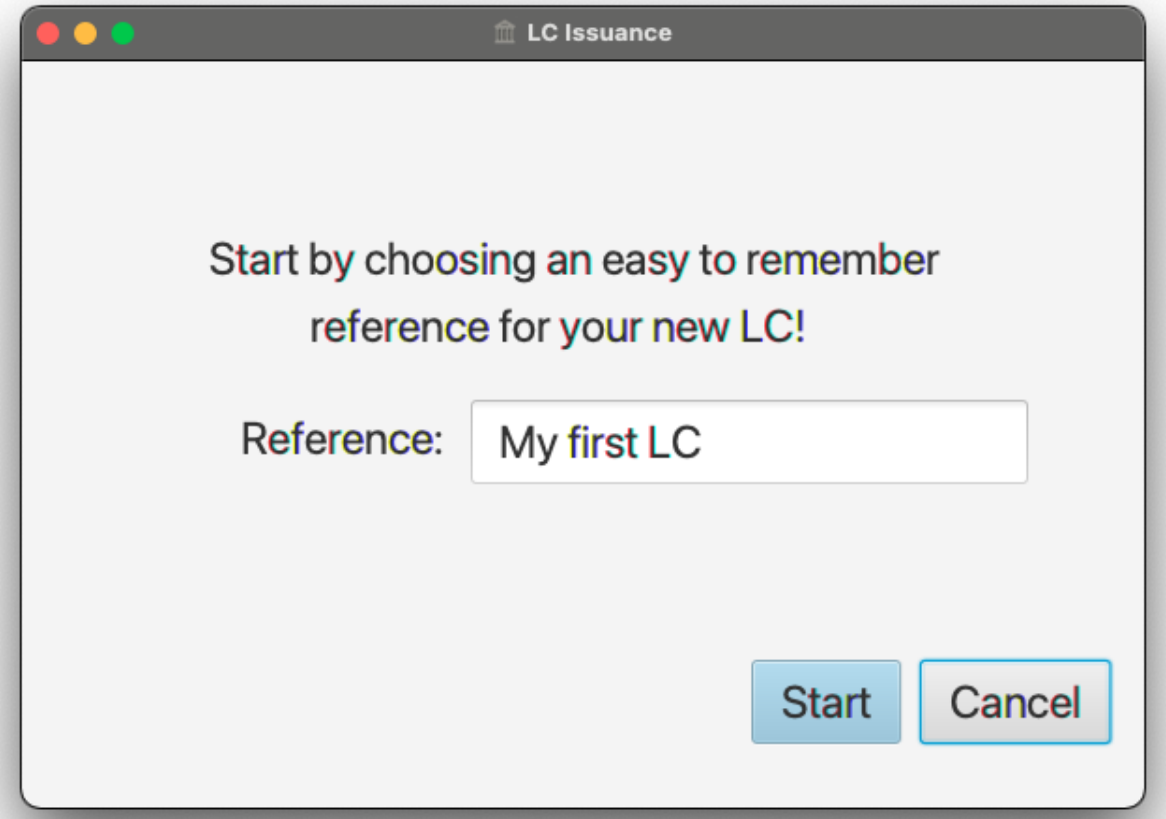

Figure 6. The start button is enabled with a valid client reference

Now that we have the start button enabling correctly, let's implement the actual creation of the LC itself by invoking the backend API.

Integrating with the backend

LC creation is a complex process, requiring information about a variety of items as evidenced in figure Revised commands when we decomposed the LC creation process. In this section, we will integrate the UI with the command to start creation of a new LC. This happens when we press the *Start* button on the Start LC creation screen. The revised `StartNewLCApplicationCommand` looks as shown here:

```
@Data
public class StartNewLCApplicationCommand {
    private final String applicantId;
    private final LCApplicationId id;
    private final String clientReference;

    private StartNewLCApplicationCommand(String applicantId, String clientReference) {
        this.id = LCApplicationId.randomId();
        this.applicantId = applicantId;
        this.clientReference = clientReference;
    }

    public static StartNewLCApplicationCommand startApplication( ①
                    String applicantId,
                    String clientReference) {
        return new StartNewLCApplicationCommand(applicantId, clientReference);
    }
}
```

① To start a new LC application, we need an `applicantId` and a `clientReference`.

Given that we are using the MVVM pattern, the code to invoke the backend service is part of the view-model. Let's test-drive this functionality:

```
@ExtendWith(MockitoExtension.class)
class StartLCViewModelTests {

    @Mock
    private BackendService service;

    @BeforeEach
    void before() {
        int clientReferenceMinLength = 4;
        viewModel = new StartLCViewModel(clientReferenceMinLength, service);
    }

    @Test
    void shouldNotInvokeBackendIfStartButtonIsDisabled() {
        viewModel.setClientReference("");
        viewModel.startNewLC();

        Mockito.verifyNoInteractions(service);
    }
}
```

The view-model is enhanced accordingly to inject an instance of the `BackendService` and looks as shown here:

```
public class StartLCViewModel implements ViewModel {

    private final BackendService service;
    // Other members omitted for brevity

    public StartLCViewModel(int clientReferenceMinLength,
                            BackendService service) {
        this.service = service;
        // Other code omitted for brevity
    }

    public void startNewLC() {
        // TODO: invoke backend!
    }
}
```

Now a test to actually make sure that the backend gets invoked only when a valid client reference is input:

```
class StartLCViewModelTests {
    // ...

    @BeforeEach
    void before() {
        viewModel = new StartLCViewModel(4, service);
        viewModel.setLoggedInUser(new LoggedInUserScope("test-applicant"));   ①
    }

    @Test
    void shouldNotInvokeBackendIfStartButtonIsDisabled() {
        viewModel.setClientReference("");
        viewModel.startNewLC();

        Mockito.verifyNoInteractions(service);                                ②
    }

    @Test
    void shouldInvokeBackendWhenStartingCreationOfNewLC() {
        viewModel.setClientReference("My first LC");
        viewModel.startNewLC();

        Mockito.verify(service).startNewLC("test-applicant", "My first LC");  ③
    }
}
```

① We set the logged in user

② When the client reference is blank, there should be no interactions with the backend service.

③ When a valid value for the client reference is entered, the backend should be invoked with the

entered value.

The implementation to make this test pass, then looks like this:

```
public class StartLCViewModel {
    //...
    public void startNewLC() {
        if (!getStartDisabled()) {                  ①
            service.startNewLC(
                    userScope.getLoggedInUserId(),
                    getClientReference());          ②
        }
    }
    //...
}
```

① We check that the start button is enabled before invoking the backend.

② The actual backend call with the appropriate values.

Now let's look at how to integrate the backend call from the view. We test this in a UI test as shown here:

```
@UITest
public class StartLCViewTests {

    @MockBean
    private BackendService service;                                 ①

    //...

    @Test
    void shouldLaunchLCDetailsWhenCreationIsSuccessful(FxRobot robot) {
        final String clientReference = "My first LC";
        LCApplicationId lcApplicationId = LCApplicationId.randomId();

        when(service.startNewLC("test-applicant", clientReference))
                .thenReturn(lcApplicationId);                       ②

        robot.lookup("#client-reference")
            .queryAs(TextField.class)
            .setText(clientReference);                              ③
        robot.clickOn("#start-button");                             ④

        Mockito.verify(service).startNewLC("test-applicant", clientReference); ⑤

        verifyThat("#lc-details-screen", isVisible());              ⑥
    }
}
```

① We inject a mock instance of the backend service.

② We stub the call to the backend to return successfully.

③ We type in a valid value for the client reference.

④ We click on the `start` button.

⑤ We verify that the service was indeed invoked with the correct arguments.

⑥ We verify that we have moved to the next screen in the UI (the LC details screen).

Let's also look at what happens when the service invocation fails in another test:

```
public class StartLCViewTests {
    //...
    @Test
    void shouldStayOnCreateLCScreenOnCreationFailure(FxRobot robot) {
        final String clientReference = "My first LC";
        when(service.startNewLC("test-applicant", clientReference))
            .thenThrow(new RuntimeException("Failed!!"));   ①

        robot.lookup("#client-reference")
            .queryAs(TextField.class)
            .setText(clientReference);
        robot.clickOn("#start-button");

        verifyThat("#start-lc-screen", isVisible());        ②
    }
}
```

① We stub the backend service call to fail with an exception.

② We verify that we continue to remain on the `start-lc-screen`.

The view implementation for this functionality is shown here:

```
import javafx.concurrent.Service;

public class StartLCView {
    //...
    public void start(ActionEvent event) {
        new Service<Void>() {                    ①
            @Override
            private Task<Void> createTask() {
                return new Task<>() {
                    @Override
                    private Void call() {
                        viewModel.startNewLC();  ②
                        return null;
                    }
                };
            }

            @Override
            private void succeeded() {
                Stage stage = UIUtils.getStage(event);
                showLCDetailsView(stage);        ③
            }

            @Override
            private void failed() {
                // Nothing for now. Remain on the same screen.
            }
        }.start();
    }
}
```

① JavaFX, like most frontend frameworks, is single-threaded and requires that long-running tasks not be invoked on the UI thread. For this purpose, it provides the `javafx.concurrent.Service` abstraction to handle such interactions elegantly in a background thread.

② The actual invocation of the backend through the view-model happens here.

③ We show the next screen to enter more LC details here.

Finally, the service implementation itself is shown here:

```
import org.axonframework.commandhandling.gateway.CommandGateway;

@Service
public class BackendService {

    private final CommandGateway gateway;                          ①

    public BackendService(CommandGateway gateway) {
        this.gateway = gateway;
    }

    public LCApplicationId startNewLC(String applicantId, String clientReference) {
        return gateway.sendAndWait(                                ②
                   startApplication(applicantId, clientReference)
               );
    }
}
```

① We inject the `org.axonframework.commandhandling.gateway.CommandGateway` provided by the Axon framework to invoke the command.

② The actual invocation of the backend using the `sendAndWait` method happens here. In this case, we are blocking until the backend call completes. There are other variations that do not require this kind of blocking. Please refer to the Axon framework documentation for more details.

We have now seen a complete example of how to implement the UI and invoke the backend API.

Summary

In this chapter, we looked the nuances of API styles and clarified why it is very important to design APIs that capture the users' intent closely. We looked at the differences between CRUD-based and task-based APIs. Finally, we implemented the UI making use of the MVVM design pattern and demonstrated how it aids in test-driving frontend functionality.

Now that we have implemented the creation of new LC, for implementing the subsequent commands we will require access to an existing LC. In the next chapter, we will look at how to implement the query side and how to keep it in sync with the command side.

Questions

- What kind of APIs do you come up with in your domain? CRUD-based? Task-based? Something else?
- How do consumers find your APIs? Do they have to implement further translations of your APIs to consume them meaningfully?
- Are you able to test-drive your front-end functionality? Do you see merit in this approach?

Further reading

Title	Author	Location
Task-drien user interfaces	Oleksandr Sukholeyster	https://www.uxmatters.com/mt/archives/2014/12/task-driven-user-interfaces.php
Business logic, a different perspective	Udi Dahan	https://vimeo.com/131757759
The Fallacies of Enterprise Computing	Ted Neward	http://blogs.tedneward.com/post/enterprise-computing-fallacies/
GUI architectures	Martin Fowler	https://martinfowler.com/eaaDev/uiArchs.html

[1] https://openjfx.com/

[2] https://en.wikipedia.org/wiki/Principle_of_least_privilege

[3] https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html

[4] https://vimeo.com/131757759

[5] http://blogs.tedneward.com/post/enterprise-computing-fallacies/