

Projektbeskrivning

CineMate: Your movie companion

2017-05-10

Projektmedlemmar:

Zino Kader <zinka766@student.liu.se>

Handledare:

Mariusz Wzorek <mariusz.wzorek@liu.se>

Table of Contents

1. Introduktion till projektet.....	2
2. Ytterligare bakgrundsinformation.....	2
3. Milstolpar	2
4. Övriga implementationsförberedelser.....	3
5. Utveckling och samarbete.....	3
6. Implementationsbeskrivning.....	Error! Bookmark not defined.
6.1. Milstolpar.....	Error! Bookmark not defined.
6.2. Dokumentation för programkod, inklusive UML-diagram	Error! Bookmark not defined.
6.3. Användning av fritt material.....	Error! Bookmark not defined.
6.4. Användning av objektorientering	Error! Bookmark not defined.
6.5. Motiverade designbeslut med alternativ	Error! Bookmark not defined.
7. Användarmanual.....	Error! Bookmark not defined.
8. Slutgiltiga betygsambitioner	Error! Bookmark not defined.
9. Utvärdering och erfarenheter	Error! Bookmark not defined.

Planering

Den här delen av projektbeskrivningen skriver ni i samband med första inlämningen, gärna under tiden ni arbetar på sista labben för att göra det möjligt att få kommentarer innan projektstart. Den kan kompletteras senare, men försök få med så mycket som möjligt redan från början.

Läs först genom <http://www.ida.liu.se/~TDDD78/labs/2017/select> om att välja projekt!

Instruktionerna nedan tar ni så klart bort när ni skrivit klart projektbeskrivningen.

1. Introduktion till projektet

CineMate is a desktop application that makes life easier for movie fanatics. Search for movies, TV-shows, actors, directors and just about anything else even loosely connected to film and get the results presented to you in a readable and digestible fashion. Create to-watch lists, top lists and save notes about movies you've seen. These are just some of the possibilities with CineMate. The information is grabbed from a reliable source/API, TheMovieDB, which supplies everything from images to movie descriptions.

2. Ytterligare bakgrundsinformation

TheMovieDB API: <https://developers.themoviedb.org/3/getting-started/introduction>

For this project, some knowledge of parsing JSON with Java is needed. I have done this before, but if anything comes in my way I'm confident in that the answers can be found in the extensive resources available online on this subject. Working with non-static resources such as sources from web APIs can introduce problems that need a lot of non-OO related work, therefore I'm going to make use of RxJava to reactively get this information and display it when it's ready and otherwise easily disregarding it. This is hopefully going to save me from a lot of headache. I've used RxJava earlier for an Android project

(<https://github.com/ZinoKader/SpotIQ>) and I feel somewhat confident in making use of it.

Moreover, to nicely present something as information-dense as movies, using a different framework would be preferred as Swing is very limited in its capabilities and does not lend itself easily to modern design patterns. This will be looked into further, but as of now a Java web framework such as Spark might be of potential interest. I'd like to discuss this further with the supervisor.

3. Milstolpar

Beskrivning	
1	Getting and setting/saving an API-key to a property file works. Asks the user for an API-key on startup.
2	Data types (models) for basic information like Movie, Series, Person and so forth are implemented and have at least some inheritance-like structure to begin with (Actor and Director are instances of superclass Person).
3	Implementation of some basic information retrieval from the API works using RxJava.
4	Getting information from the API and setting it to the correspondig models work. This is demonstrated in a test class which creates instances of models (Actor, Director etc...), sets the information for these properly from the grabbed JSON data and prints the information out nicely using the models' customized overridden toString() methods.
5	Search queries work in the UI and feeds the user with the results in a readable manner (i.e. a list of movies for a movie query).
6	Add fully working login option that lets the user save favorites lists and watch lists etc...
7	More milestones will be added as the project proceeds...

4. Övriga implementationsförberedelser

This project has a lot of potential for displaying OO-programming capabilities. Implementation details such as superclasses and their inheriters are easy to visualize with data such as Person, Actor, Director where the obvious choice here is to have Actor and Director extend the more abstract and generalized Person. The goal here is to make extensive use of polymorphism to convert between models as little as possible and instead freely pass around these data in the program with little concern. Furthermore, the API information will be exposed as observables and subscribed to by observers. The UI framework is still undecided and **I'd like some suggestions** when it comes to this. Have students previously made use of Java web frameworks such as Spark successfully, or has it been too much of a hassle?

5. Utveckling och samarbete

I will be working solo on this project and therefore there is not a lot to discuss here. As a note to myself, I'm aiming for a 5 in this course and therefore the project is highly prioritized over other work in other courses.

Implementation and final project description

Milestones

The milestones were unfortunately not updated as this was a one-man project and I kept it all in my head instead. The very basic milestones are all implemented, except for favorite- and watch-lists, which I felt were superfluous and required a lot of non-OO work.

Code documentation overview

The MVC pattern

The MVC pattern and its implementation described below is considered as **design decision 3**.

The project makes use of the MVC (Model-View-Controller) design pattern to delegate tasks to appropriate classes. As such, the program is divided into models, views and a controller. A brief explanation of the pattern is that the controller is user-facing, it handles and delegates user input, formulates a new change in the application state by for instance manipulating a model (e.g. changing the date), which is then sent to the view for rendering. So, if we look at this example from the user's point of view for a hypothetical calendar application, the user would be looking at the UI which shows the calendar appointments for today. Say that the user changes the date to display, we would now have to capture this interaction in our controller, get the new date which the user has requested to show, send this date to our model layer which will figure out an appropriate model to send back to us, and then send this to the view layer for rendering.

To further explain how I've made use of the pattern in my application see the provided image below.

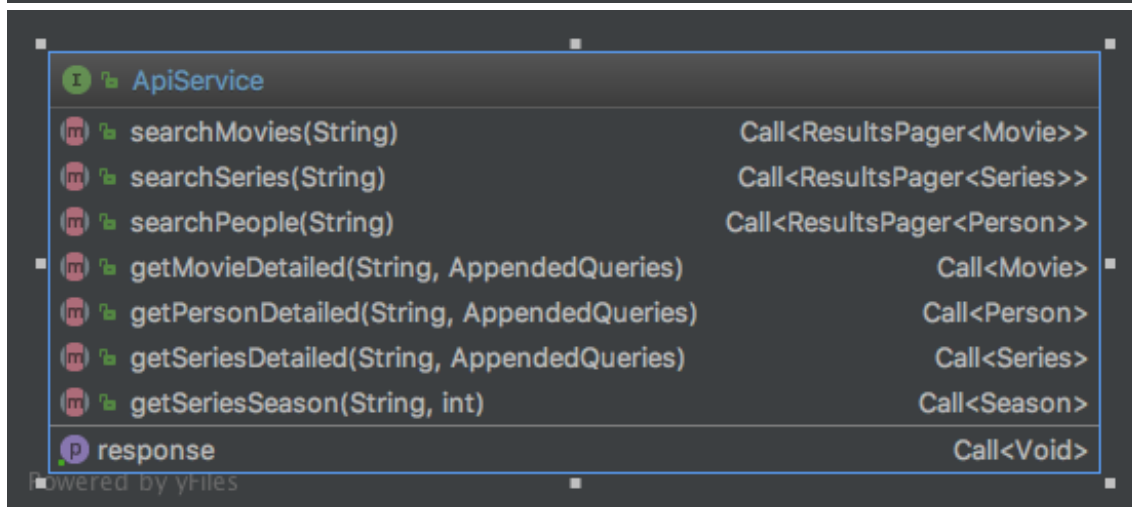
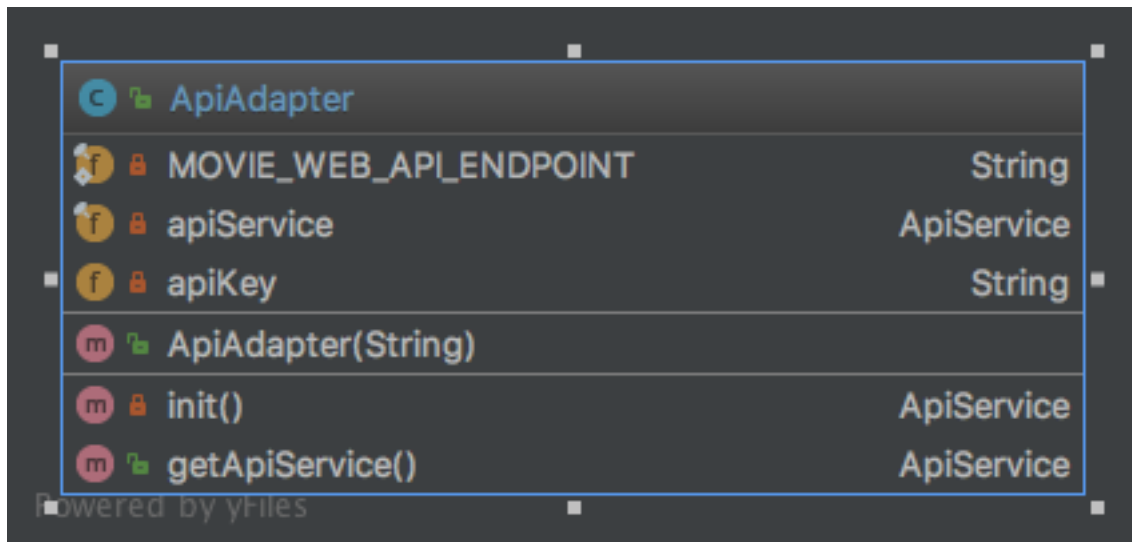
```
@FXML
public void handleCastClicked(MouseEvent mouseEvent) {
    if(mouseEvent.getClickCount() == FXConstants.DOUBLE_CLICK_COUNT) {
        Cast selectedCast = castListView.getSelectionModel().getSelectedItem();
        screenParent.loadWindow(CineMateApplication.PERSON_WINDOW_FXML, selectedCast);
        closeWindow();
    }
}
```

Here we have a listener that listens for clicks on any of the cast members' pictures. This method is found in our **controller**, which handles all the user input. When one of the cast members are double-clicked, we get the appropriate **model** (by figuring out where the user clicked), pack it up in the Cast model, and send it away to the **view** by loading up a new window with the information we just got from the user input.

How we get, serialize and use data

The project makes use of a number of libraries that makes networking and JSON processing easier, for instance Retrofit, Gson and OkHttp to name a few. To start getting data from our endpoint, TMDb (The Movie Database), we create an ApiAdapter by passing in our API key to the constructor and saving it in the apiKey field. We then create an ApiService from our constructor method that uses our API key in its implementation, and we immediately save our custom implementation of our ApiService to the field apiService. Our custom implementation of ApiService is created using the init() method in ApiAdapter, here we make some adjustments, for example the API key will be appended to all queries that we send to ApiService automatically (line 42-48 in ApiAdapter).

To use our ApiService to get data from the API, we now ask the ApiAdapter to send us back our specialized ApiService using getApiService(). Now we can use all of our methods defined in ApiService to get data from the API. We can for instance search for people, movie, series, series seasons (and episodes) and so on.



To further make the implementation of the TMDB API clear, an in-application use is provided in the image below.

```
public void handleSubmitApiKey() {  
    String apiKey = apiKeyTextField.getText();  
  
    ApiAdapter apiAdapter = new ApiAdapter(apiKey);  
    ApiService apiService = apiAdapter.getApiService();  
  
    apiService.getResponse().enqueue(new Callback<Void>() {
```

This is how we validate whether the API key is correct or not. The method can be found in StartupScreenController, line 110.

What we do is get the submitted API key from the text field which the user has entered it into. We then proceed to create our ApiAdapter, passing the API key into its constructor method, so that ApiAdapter can customize our ApiService. We then assign our newly created ApiService field to the one returned by our ApiAdapter using apiAdapter.getApiService().

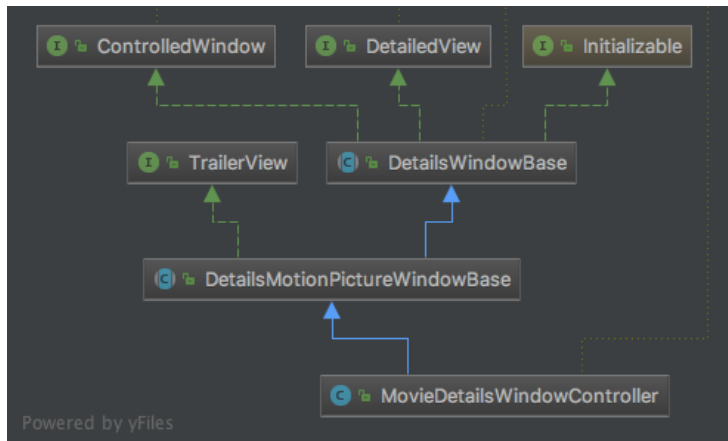
We can now proceed to use the getResponse() method from our ApiService, this will allow us to check the API key against the TMDB servers and validate its legitimacy.

How controllers (screens, windows) are managed

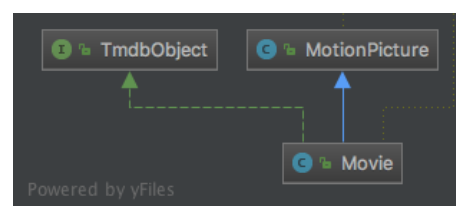
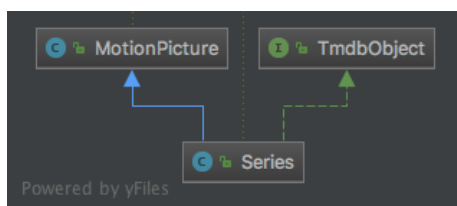
CineMate starts up in the CinemateApplication class and proceeds to create a ScreenController which holds screens and windows and loads them up when requested to. We add all of our used screens to it along with information needed for ScreenController to properly display them.

We then use ScreenController throughout our application to load, hide, close, show and everything else that has to do with screens.

To demonstrate how we've abstracted our screens and windows see the image below which displays the inheritance chain of MovieDetailsWindowController.



As can be seen, our `MovieDetailsWindowController` is a `DetailsMotionPictureWindowBase`. Both `MovieDetailsWindowController` and `SeriesDetailsWindowController` (from here on referred to as “mentioned subclasses”) are `DetailsMotionPictureWindowBases`, as this class contains methods and abstractions specific to windows that deal with `MotionPicture`-type objects, which both of the mentioned subclasses do (`Movie` and `Series` objects both extend `MotionPicture`).

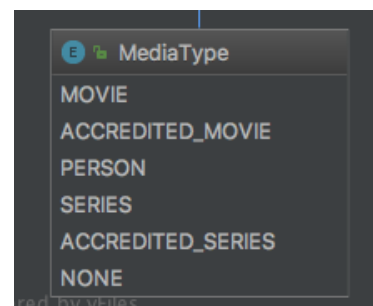
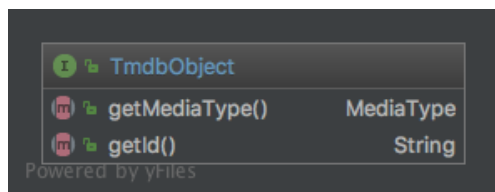


Furthermore, since both of the mentioned subclasses need to show trailers in their windows, they both implement the `TrailerView` interface. To avoid implementing `TrailerView` twice, once in each subclass, we instead implement it in `DetailsMotionPictureWindowBase`, and as our subclasses are just that, subclasses of `DetailsMotionPictureWindowBase`, they will inherit the implementation of `TrailerView` from their superclass.

To continue on the subject of only writing shared code once, we take a look at `DetailsMotionPictureWindowBase`, and it's apparent that it is in itself a subclass to `DetailsWindowBase`. This master-class for windows contain the most basic implementations needed to create a window which isn't in any way specialized. It implements a couple of interfaces itself, mainly our view interface for windows, `DetailedView`. Here, we have methods that windows need to get going with showing information, and these methods are inherited all the way down from `DetailsWindowBase` to `MovieDetailsWindowController`, where we override the methods and provide our own custom implementation for it for our specific use-case. This is the beauty of inheritance that object-oriented programming languages provide us with. We have subclasses of subclasses of subclasses, which decrease in level of abstraction as you go further down in the inheritance chain.

A brief explanation of the models

To get a hang of how we handle models in this application, we'll look at the most basic model, `TmdbObject` and how it is used. The first three models created for the application were `Movie`, `Series` and `Person`, these are the most basic things we want to search for within the application. There is a dilemma, though, should all three of these extend the same superclass when they aren't all that much alike? Well, we want to collect these models under one superclass so we can make use of polymorphism to easily pass them around in the same `List<Movie/Series/Person>`, for instance. This is crucial for our program to work in the most basic way. So, `Movies` and `Series` are very much alike as they share a lot of fields and functionality and really, you can hear it on the name, they're both something you can watch, they're both motion-pictures. And so, the `MotionPicture` superclass was created to abstract down the shared functionality between `Movie` and `Series`. Our `Person` model can't extend this class because it has next to none of the shared functionality. Instead, we create an interface called `TmdbObject`, which has a method for getting the media type and a method for getting the ID of the object, see images below.



Since `Movie`, `Series` and `Person` implement this interface, they are all considered to be `TmdbObjects`. We can now pass movies, series and persons in a `List<TmdbObject>`, and later when we need to have them in their less abstract forms (as `Movies`, `Series` or `Persons`), we can simply cast them correctly by getting the `MediaType` of the `TmdbObject`, and then correctly casting the `TmdbObject` based on which `MediaType` it had.

An implementation in our program which makes use of all of these abstractions can be seen in the image below.

```
switch (mediaType) {  
    case MOVIE:  
        List<Movie> movies = tmdbObjects.stream()  
            .filter(tmdbItem -> tmdbItem.getMediaType().equals(MediaType.MOVIE))  
            .map(tmdbItem -> (Movie) tmdbItem).collect(Collectors.toList());  
}
```

The switch-case and its implementation described below is considered as **design decision 5**.

Here we run a switch on the MediaType of our TmdbObject, and in our first instance, if it happens to be a Movie, we create a List<Movie> from our “tmdbObjects” field and filter for TmdbObjects that are of the MediaType “MOVIE”. In the very same method (populateList() in MainScreenController), we run the same operation for Series and Person cases, and our input data was just a List of TmdbObjects, nothing else. This demonstrates how abstractions like this can make life easier when adding new features. For instance, if we were to add a new model called Example, which implements TmdbObject, nothing would break because our implementation figures out which MediaType it has before doing anything to it. We could safely pass in a List<TmdbObject> which contains any number of Example objects, and in our method implementation, they would simply be ignored because they don’t match any of the MediaTypes we filter for.

Use of free material and libraries

To accomplish everything from networking and JSON parsing to good UI design in a feasible amount of time, to lay my focus towards OO-programming instead, I've made use of a few libraries listed below.

- TMDb API: API used for fetching film metadata
- Retrofit: REST client used to create a Java interface for API calls
- Gson: Used for converting/parsing JSON
- OkHttp: HTTP client which Retrofit is built upon
- JavaFX: The platform for creating the desktop application
- ControlsFX: Extends some UI controls for JavaFX
- JFoenix: Material design styles for JavaFX
- MinLog: Small logging library for easier and prettier logging

Design decisions

1. I am using streams where iterating over items and adding them to a list based on certain conditions can be simplified to a stream of items, which are filtered based on a condition and then collected into a final list. Examples of this can be seen in `MainScreenController` and `CrewHelper`. I'm using streams primarily because they are more concise, often turning 10 lines into a single line, and less code is just fewer places where errors can occur. An example of what could be done instead of using streams is shown in `CrewHelper`'s `filterDirector()` method.
2. When dealing with images, primarily in `ImageCache`, we delegate each image fetching operation to a new thread. This makes use of multithreading if supported on the machine and lets us create dozens of threads to load up several images concurrently. This is a critical solution for an image-intensive app like `CineMate` that displays a lot of images in large lists, such as the search page. As such, loading our images on one thread would've been an option, although not a good one.
3. I've chosen to make use of the MVC-pattern for the project structure. This has greatly simplified debugging and testing and made working with the project much easier. For instance, adding new models and features in general doesn't affect the rest of the code as it's decoupled by definition of the MVC pattern. Of course, the program could have made use of any of the dozen of other design patterns instead, and functionality would have been the same had they been implemented correctly. MVC was used because I found it fitting for our purposes. More about this design decision can be found under the headline **The MVC pattern**.
4. `Episode` and `Season` both implement `SeriesDetail` to be able to be passed in to the same `Cell`. This will save us the creation of another `Cell` type class. By use of polymorphism, these data types are considered the same as long as they are abstracted as a `SeriesDetail`. Later, we make use of the `SeriesDetail`'s method `getType()` to determine the `TvType` of our object, and from there on we know whether to cast it to an `Episode` or a `Season`.
5. Found under the chapter **A brief explanation of the models** at the bottom of the chapter, find it by searching for "design decision 5".
6. All images in the application are cached after use. This is done with the help of a `Map` of the images wrapped in a `SoftReference`. I chose to implement caching because the application loads a lot of images in one search, and would unload them from memory when they moved out of the user's view when the user would scroll down a list. I wanted to keep them accessible for cheap when the user for instance scrolls up again to the same images that were loaded earlier. To keep the memory from getting too full, `SoftReference` was used which tells the garbage collector that it's quite alright to gc the reference to the image if memory is low. The decision to implement image caching did great things for the performance of the application, which was sluggish before.
7. By making use of abstract class properties in `DetailsWindowBase` and `DetailsMotionPictureWindowBase`, we can for instance call interface methods in our abstract class, and be completely sure that subclasses that extend `DetailsWindowBase` and `DetailsMotionPictureWindowBase` have implementations of these methods. This is also explained in the javadocs for both of the aforementioned abstract classes. Instead of doing this, we could've left our current abstract superclasses as concrete classes and called the inherited interface methods from our subclasses, once for each subclass, which would have worked, but that would've been repeating of code that can be generalized and only written once.
8. All images that have text displayed over them in the application are analyzed using `TextColorHelper` and its only method `setContentAwareTextColor()`, to figure out which text color to use for the text on top of the image. Since our application displays images of all colors and shapes, having a final, unadapting text color is a bad option because the text color might have the same general brightness as the background image that it is sitting upon. A good example of this is the backdrop for the series "How I Met Your Mother", which is almost entirely white. Having a white text color for the series title on top of a white image would render the text completely invisible. There are some obvious cons with this solution. The biggest con is the performance hit. We need to analyze sometimes several thousand pixels wide and high images, and summarize the average color for these. This is an expensive operation performance-wise and the performance hit is visible when turning the feature on and off. After some thought, I decided to leave the feature in as the pros outweighed the cons. I would happily wait an extra second for a movie to load, and be able to read the title, description, rating and other metadata, rather than it loading faster with hardly visible text.

Use of object oriented patterns

1 and 2. **Use of abstract classes and interfaces.** As described in **How controllers (screens, windows) are managed**, we make use of abstract classes to specify common functionality between several classes. An example of this is the `DetailsWindowBase`, which is an abstract class that contains some abstract implementations for basic window functionality, this class is a superclass to another abstract class, `DetailsMotionPictureWindowBase`, which contains less abstract implementations and functionality for a more specialized type of window that handles `MotionPictures`. And again, this class is a superclass of for instance `MovieDetailsWindowController`, which has inherited the implementations of all the aforementioned classes. Abstract classes are useful for “abstracting away” functionality under several layers, to have less specific and more specific classes in terms of functionality. This makes changes to a superclass with lots of subclasses easy, as they all inherit the functionality that you’re making changes to, you only need to make the change in the abstract superclass, and it will apply to all of its subclasses. There are also more benefits to using abstract classes. For instance, all the interfaces that an abstract class implement, can be overridden and implemented in the abstract class, or left to the subclasses for implementation. This means that you don’t need to implement the interfaces in your abstract class if it doesn’t make sense to, but you can “make sure” that all the subclasses of your abstract class have an implementation of the interface, because they must according to the contract. If we chose to leave our abstract class concrete instead, we would have to implement the interfaces in the concrete superclass instead, and leave the implemented functions empty if we didn’t need them implemented in this abstraction layer yet, which looks as bad as it sounds. Furthermore, as we use an abstract class and can “make sure” our subclasses have inherited and implemented the interfaces implemented by our abstract class, we can call methods of the interface straight from our abstract class methods, and they will be applied to and run on the subclasses to the abstract class. An example of this can be found in `DetailsWindowBase`, line 78, where we call `delegateSetData()` which is a function that `DetailsWindowBase` implements, but doesn’t override. So, the implemented function will instead run on the overridden implementations of its subclasses, for instance `MovieDetailsWindowController` will have its overridden `delegateSetData()` method run on startup.

3. **Implementation inheritance.** All of our models are extending other concrete models when it makes sense to. Examples of a long chain of implementation inheritance can be found with `AccreditedMovie`, which is a `Movie`, which itself is a `MotionPicture` by subtyping polymorphism. All of these are concrete concrete classes and the power of overriding can be seen in how `MotionPicture` has a `mediaType` field which holds the `MediaType` of the subclass. In `MotionPicture`, all it does is return the `mediaType`, which is set automatically by JSON serializing, but in its subclasses, for instance `Movie`, we override the `getMediaType()` method that is inherited from `MotionPicture`, and we return `MediaType.MOVIE` instead. The same functionality applies to `Series`.

4. **Method overloading.** We use method overloading mainly to provide the person working with the project many options when calling a method. An example of method overloading can be seen in `MessageHelper` and its `showMessage()` methods. The default method has one parameter, *String message*, while there is another method which has the first parameter from the default method, but also a second parameter, *int duration*. The default method uses a constant *duration* when you pass it a message, while the overloaded method allows you to specify your own duration. This is useful because most of the messages shown to the user in my application only need the default duration which is currently 2500ms. But every once in a while, you might want to specify your own duration for an extra-important message that the user *really* needs to pay attention to. This could be accomplished by naming the method something else in a non-OO implementation, for instance we have the default method `showMessage()` and another method which allows custom duration called `showMessageFor()`.

5. **Encapsulation.** With all of our models, really, we are using encapsulation to get our private fields. The fields cannot be accessed directly by calling `model.myfield`, instead we call `model.getField()`. This allows us to control how the programmer gets and sets the data. For instance, we have chosen in all of our DAO that are derived from JSON parsing that you

cannot set any fields for it. The fields are private and there are no setters, so all you can do is get them. This is because we should not be setting our own data on the models that we get from the API, they should be complete already. But we do make use of the extra control for the getters. An example of this can be found in `Series`, for instance. The field `episodeRunTimes` is a list of integers that serializes straight from the API. In raw, it looks like `{5500, 4800, 6120}` and so on, but the raw data is never accessible to the programmer dealing with the `Series` model. The raw data is a list of the amount of seconds that the different episodes of the series run for, this is not something the programmer needs in specific for the application. If we want to get the runtime somehow, the model only allows us to get the average runtime of all the episodes in the series, which is done by calling `getRunTime()`. By encapsulation, we've protected our fields, collected them into a digestible model, and controlled how they are implemented, all from inside the model class.

User manual

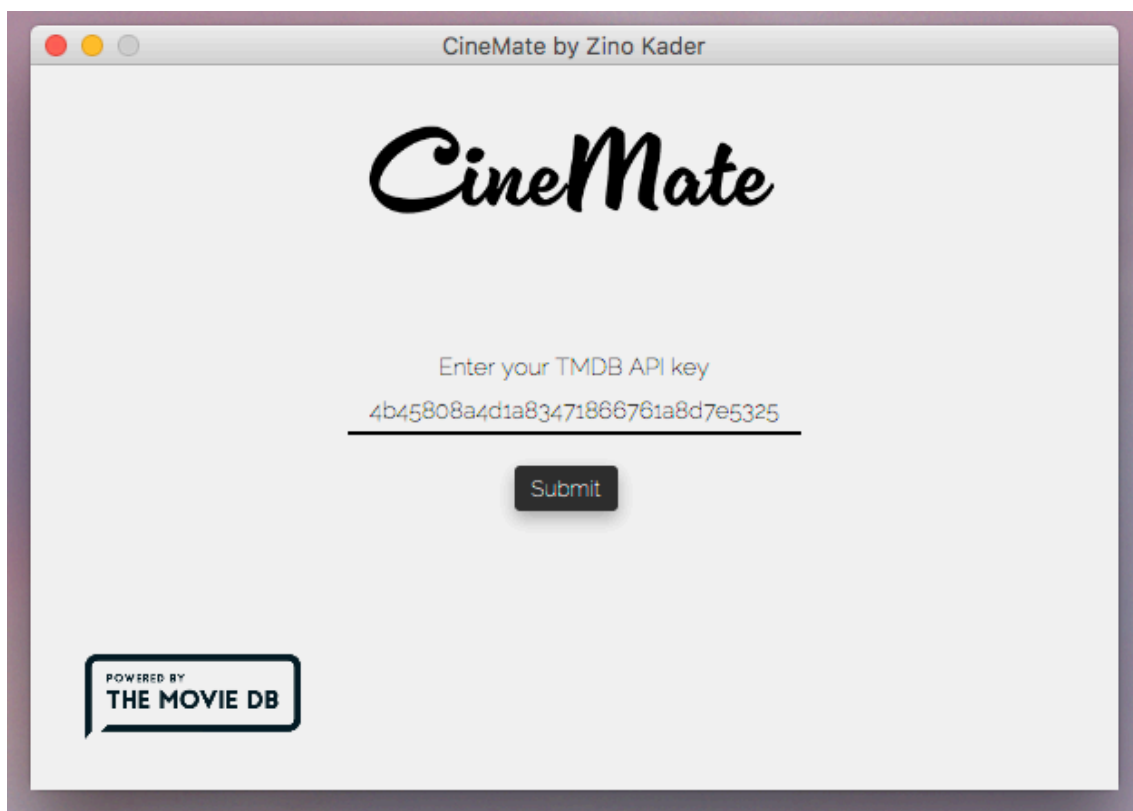
TMDB API key: 4b45808a4d1a83471866761a8d7e5325

The program is very straight-forward to use. Keep in mind that most faces of people and film posters can be clicked throughout the application, so try clicking anything and everything!

To start off, I've left the API key in the application automatically filled in. If you want to test how it handles wrong API keys, disable this automatic login by commenting out the `setText()` method in line 76 in class `StartupScreenController`. Image provided below.

```
//comment this out to not login automatically  
Platform.runLater(this::delayLogin);
```

When starting the application, you're presented with an interface as such:



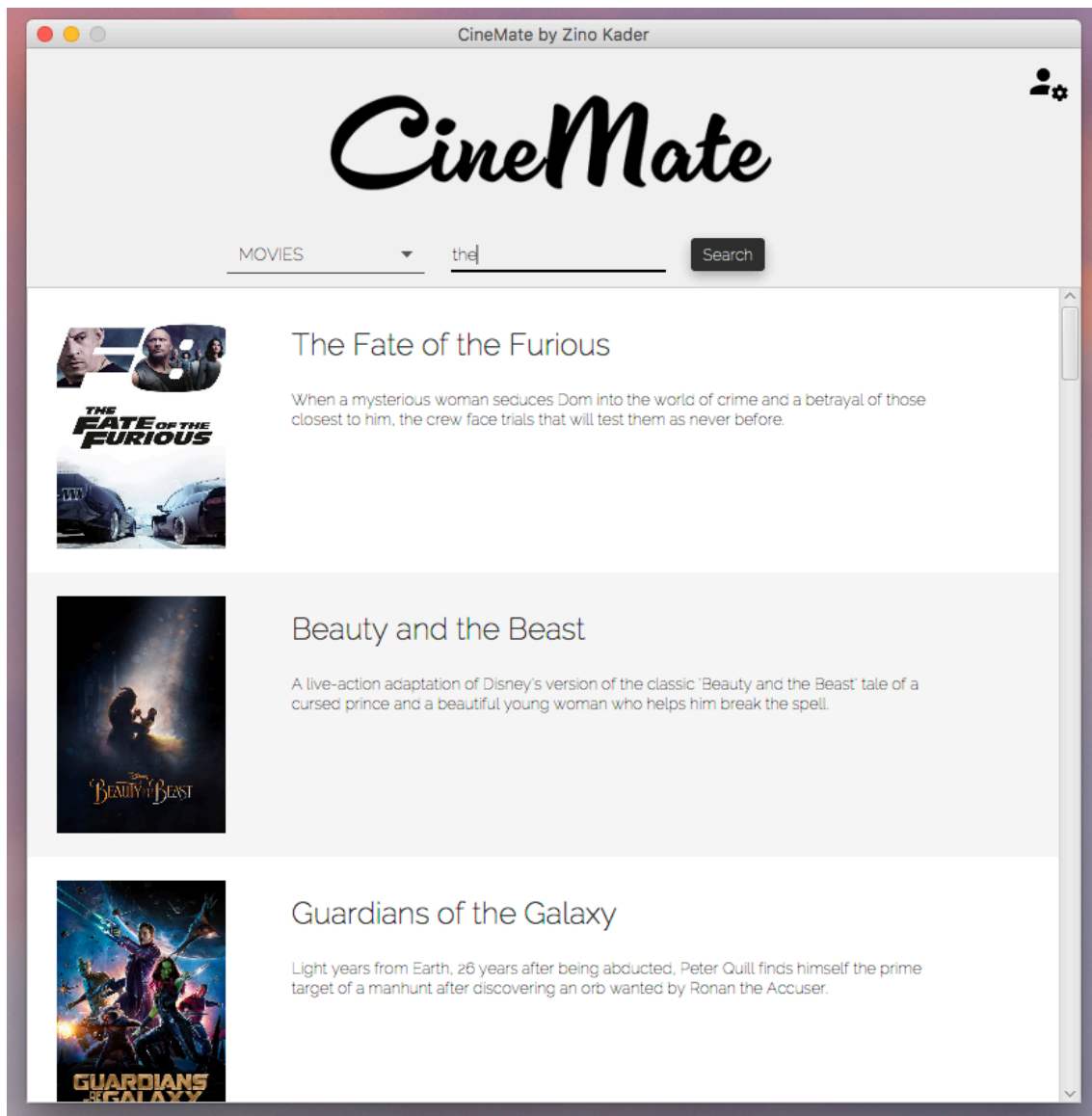
Enter the API key, try removing some characters to see how it handles errors, and finally press submit with the correct API key to proceed.

Next, we're going to take a look at the main window. When successfully logged in, you will be presented with this window:



Click the dropdown that is already dropped-down in the image and choose what you want to search for. I recommend selecting movies and searching for the phrase "the" to get movies that most likely are blockbusters and have information available for the API to grab. You can press the "Search" button or press enter to search.

Your search should be looking like this:



Now, you can **double-click** any of the results to open them up in a new window with many more details. We'll go with "Guardians of the Galaxy" for this example.

It should be looking like this:

Guardians of the Galaxy

Guardians of the Galaxy

★★★★☆


2014-07-3002:01:00

Budget: SEK170,000,000.00


Revenue: SEK773,328,629.00

Light years from Earth, 26 years after being abducted, Peter Quill finds himself the prime target of a manhunt after discovering an orb wanted by Ronan the Accuser.

Marvel's Guardians of the Galaxy - Trailer 2 (OFFICIAL)

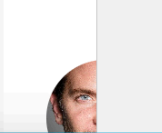

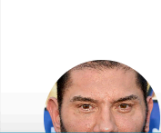
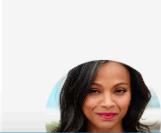



Director

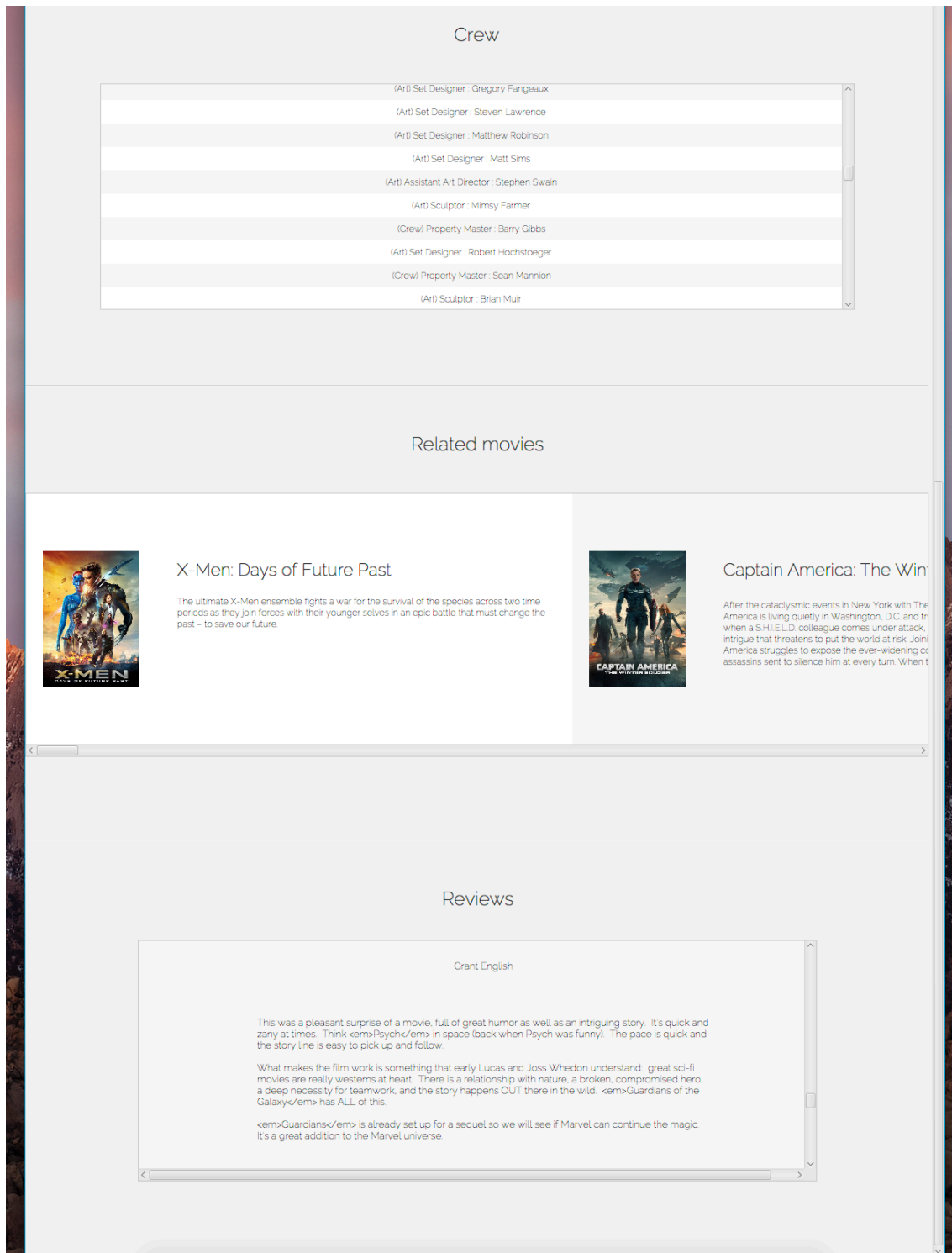


James Gunn

Cast



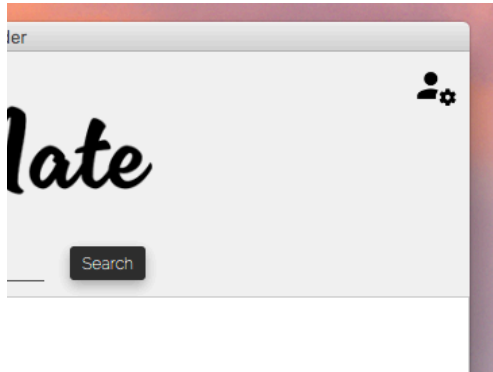
You can **double-click** the cast members in the bottom of the image to open the actors up in a new window. You can also press the trailer to view it from inside of the application. Hold your mouse over the star ratings and wait for a second and you'll see the exact rating out of 5, decimals and all! Scroll down for more clickable stuff!



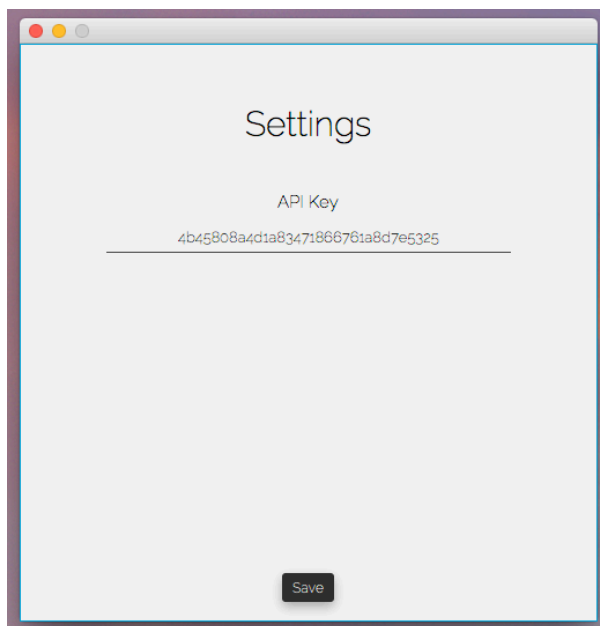
Here you can **double-click** the movies to open them up in a new window.

Back to the main screen, we would like to change our API key without restarting the application, and oh, it logs us in automatically anyway so we don't really have the time to paste in the new one and submit before it does, right?

Changing the API key can be done by pressing the settings cogwheel in the top right corner of the application in the main screen, image provided below.



Opening it up will look like this:



From here on, I think the rest can be figured out by curiosity. That's really all the basic functionality that the application has. Have fun!

Final grade ambitions

With regards to the use of modern technologies and libraries that are used to receive data from an API endpoint, I consider this project very up-to-date with what could be asked of a software engineer in this day and age. Furthermore, making use of current design architectures such as MVC (Model-View-Controller) displays an understanding of the importance of good design when creating large-scale applications. Good programming practices are used throughout the application, and code is written in the most concise way possible, for instance, streams are used where for-loops could be used instead. For these reasons in summarization, I will be aiming for a 5 in this course and for this project.

Evaluation and experiences

What went well, what didn't go so well?

From the get-go, I was sure of the design pattern to use, so structuring of the program was never an issue and I didn't have to make any major refactors of classes throughout the development of the program, which felt really good. Something that didn't go well was serialization. I oftentimes had to write code twice for classes with the same superclass which had the same field, simply because the JSON field from the TMDB API had different names for the different classes. An example of this can be seen with Movie and Series. For Movie, the title of the movie is called "title" in the JSON response, yet for Series, the title of the series is called "name" in the JSON response. This means I have to annotate the fields differently for each class, and can't move the title field to MotionPicture, which is the superclass that they both share. Maybe this could have been solved with some extra skills in Gson and serialization in general, but I also feel like it was a bad decision by the developers of the TMDB API.

Did you spend too much time / too little time?

I feel like I spent the right amount of time, and maybe a little bit too much to really make sure the code looks good. I learned a lot in the process and I'm confident that the development of this program has made me a much better developer.

What tips would you like to give future students of the course?

One really important thing when dealing with applications of this scale is design pattern. Decide on one early and stick to it. MVC is probably the oldest one known to man and there is a lot of resources online to help you understand how to structure a program with this pattern in mind. Furthermore, make notes of design choices during the development of the program. I did and therefore I had no problems when writing that part of the report.

Thanks for reading,

Zino Kader