Landsbankinn

INTERNAL CREDIT ASSESSMENT SYSTEM FOR LANDSBANKINN:

DEVELOPMENT HANDBOOK

MAY 11th 2018

SYSTEM DEVELOPED IN ASSOCIATION WITH REYKJAVÍK UNIVERSITY'S DEPARTMENT OF COMPUTER SCIENCE

AXEL BJÖRNSSON 170395-3369

DARRI VALGARÐSSON 251295-2989

EDDA STEINUNN RÚNARSDÓTTIR 241095-2909

Table of Contents

1. The Credit Assessment System

Credit assessment is a process which is often a prerequisite to successful loan applications for customers. The process involves sequences of calculations that are made based on applicants' financial position, for example on applicants' incomes, expenses and on prior loan information. From these calculations bank employees can effectively assess applicants' financial position and their capability for further expenses and thus the loan amount the bank can potentially grant to the applicant in question.

The Credit Assessment System is an internal system developed for Landsbankinn in cooperation with Reykjavík University and is designed to facilitate, automate and efficiently conduct credit assessment of applicants. The system is developed for internal use in Landsbankinn for bank employees, specifically those that handle credit assessment and/or manage loans for customers. An internal system using the same internal API as the Credit Assessment System with similar functionality had previously been deployed for internal usage, but had poor user experience, quite unsatisfying interface and users commonly encountered various unhandled errors in the system all of which made the prior system difficult to use on daily basis.

The new Credit Assessment System is yet to be deployed and is currently only available in development mode. The system depends on newly introduced technologies and is fundamentally different in structure and environment from the of the old system and all its predecessors. The old system has taken roots and dominated internal credit assessment processes internal to the bank for decades. Therefore it is important for the new Credit Assessment System's advance and viability that future maintainers of the new Credit Assessment System understand its design and implementation, especially since the original system authors will most likely not be part of future deployment or maintainance. The following handbook attempts to clarify attributes of the system in terms of it's inner environment, structure and technologies to facilitate future management of the system.

2. Installation and running the Credit Assessment System

As stated the system is currently in development mode and is yet to be deployed. Therefore the system must be installed and run on a local machine to be used. Access to project source code is restricted due to privacy policies, but currently available to stakeholders, certain bank employees and system authors. This is due to the fact that the new Credit Assessment System is not to be deployed as an independent system but as a subsystem of a large inner system containing a network of the bank's other inner systems. Once this super system and other subsystems have been designed and developed the new Credit Assessment System is adjusted and deployed as a subsystem.

The system was developed using yarn as a dependency management system instead of the commonly chosen npm due to its superiority in efficiency and speed. It is therefore is preferred to install the system's dependencies, although npm will work as well. In order to do this, one must have npm and yarn installed as well as NodeJS; all of which can be downloaded directly from https://nodejs.org/en/download/ or installed directly in the terminal on a linux based OS. Once that prerequisite has been met, yarn install can be run (installation via npm or node would work as well) to install the system's dependencies. See example terminal command below:

```
username$ pwd
~/.../liru-credit-assessment/
username$ yarn install
```

Once dependencies have been installed, the system can be run. To run the system, one must make sure one has navigated to the correct folder, then run the project via yarn start (running the project via npm or node would work as well). See example terminal command below:

```
username$ pwd
~/.../liru-credit-assessment/
username$ yarn start
```

Once prerequisites are met the project should be running at http://localhost:8080/

NOTE: Developer might encounter limitations running this project using beta versions of node or versions other than version 8 of node, which is the version currently recommended by node's maintainers. This is due to deprications of features in development tools/dependencies.

3. Usage Prerequisites

Resources are retrieved via a REST API that uses stateless HTTP requests to serve resources and update internal resources. The API's usage is prohibited if used not internally to the Bank and if user is not authenticated. Therefore, in order to obtain Credit Assessment System's full functionality the system must be run under the following circumstances:

- The system must be run inside the Landsbankinn network.
- The system must be running on a computer that is authenticated and registered on the Landsbankinn domain.
- All users of the system must be logged into the Landsbankinn domain and be connected on the company's internal network. All users must be registered as a staff member.

Since these requirements cannot always be met, a mock server was generated to serve mock resources and mimic the behavior of the bank's internal API without serving any personal information in order to use the system without meeting these requirements. See details of capabilities and limitations of the mock server under section 9 of this handbook. To run the system in such a way that it uses this mock server instead of the API with no connection into Landsbankinn internal services required the user may use the command yarn start**:dev**.

See example command to run mock server below:

```
username$ pwd
~/.../liru-credit-assessment/
username$ yarn start:dev
```

NOTE: The source code of this project is owned by Landsbankinn. Access to it is restricted due to privacy policies

If you happen have access to source code without having been explicitly granted access rights to it either by system authors or certified employees of Landsbankinn you are by definition unauthorized to access it. In such case you are encouraged to immediately forfeit your access rights in compliance to privacy policies and contact either system authors or Landsbankinn.

4. Code Structure Explained

4.1. Source Code Structure and Classification of Components

The root folder of the project consists of various subdirectories and configuration files that have different roles for the project. For this sections, all file paths shown are paths relative to the project root directory.

4.1.1 Root folder

The configuration files are all top-level, located at project root:

- ./package.json: Specifies all external project dependencies, along with scripts used to run the project and tests
- ./.travis.yml: Configuration file for a Continuous Integration service, TravisCI
- ./tsconfig.json: Configuration file used by TypeScript
- ./tsconfig.test.json: Configuration file used by the project tests in TypeScript
- ./tslint.json: The linter configuration file for TypeScript, enforces coding conventions described in section six of this report
- ./webpack.config.js: Configuration file for Webpack, which is used to run the project, both in development mode, using webpack-dev-server, and in production mode, by running *yarn build*.

The root contains other subdirectories, mainly the __tests__ folder and src folder where the project itself is separated into subfolders within the root of the project.

4.1.2 The Source Folder

The src folder, commonly referred to as the source folder, contains all project components. Each component is separated into subfolders based on which categories they fall into. The only file in the root of the src folder is the *index.tsx* file which serves as the main entry point by react into the application. The components are classified into categories as follows:

- ./src/actions: Functions or procedures used by reducers to manipulate some value or perform tasks, storing some payload obtained by the action into to a shared redux store that can be accessed by all components in system.
- ./src/constants: All system-wide constants. Mostly constants that represent actions for procedures in the actions folder.
- ./src/models: All files that represent data structures that the project uses. Should use the ES6 classes or interfaces to declare the data structures.
- ./src/reducers: Combine associated actions into a single reducer which then makes up the shared redux store.
- ./src/services: All files that represent the service layer of the project or major business logic related to the projected.
- ./src/styles: All files that contain some style information to be associated with JSX elements.
- ./src/ui: All components that are not specifically integrated for this project, i.e. could be used for another standalone project.
- ./src/view: All project specific frontend components. This is most of the systems files and logic.

4.1.3 Unit Testing Directory

The __tests__ folder contains all the unit tests written with jest. They test individual methods within each component in the src folder. See section 6.3. for a detailed explanation of the unit tests.

4.2. Documentation

All classes are preceded by a multiline comment briefly explaining any props that the class takes in and for what reason and states component keeps (for stateful components) along with brief description on the class itself and its purpose for the system. Example below is taken from source code of such example:

```
/**
* Renders the search and contains all business logic for search bar and result table
* Takes in SearchProps as props containing
*    getService: function that belongs to Redux and gets authenticated token to use API
*    history: browser history of user's session
*    service: class instance served by Redux allowing abstracted calls to API
* Takes in SearchState as props containing:
*    customers: all customers returned by search
*    isSearching: boolean, indicates if searching is in progress
*/
class Search extends React.Component <SearchProps, SearchState> { /* CLASS BODY */ }
```

All functions are preceded by a multiline comment describing its purpose, its parameters and its return value as follows in accordance with **TypeDoc** standards (see more on standards: *http://typedoc.org/guides/doccomments/*). An example of this syntax-like documentation is shown below and is an example taken from source code of a function component (dumb component):

```
/**
* Renders a remove icon with an on-click functionality of deleting the expense associated with it
* @param props, containing:
*  deleteExpense function: deletes parameter expense from API
*  expense: the expense object to be potentially deleted from API
* @returns JSX.Element of a remove icon which on click deletes expense associated with it
*/
const RemoveExpense = (props: { deleteExpense, expense }): JSX.Element => {

        /* FUNCTION BODY */

}
```

4.3. Usage of Interfaces

TypeScript Interfaces are used for components that take in required props to explicitly declare props and their types for components. Not only has implementation of interfaces proven to be powerful as documentational tool due to their nature of explicitly stating component's states and props and their types; since compiler errors are raised if programmer is to provide component with incorrect props or try to manipulate non-existent state variables, this also proved to be a vital compile-time check for errors and bugs.

Therefore interfaces are used whenever possible in system and authors highly recommend usage of interfaces to future maintainers for the system for reducing possible run-time errors. Interfaces are documented for all components with required props that indicate the purpose of state variables and props. See example below:

```
interface MonthPickerProps {
 /* the initial/placeholder date for monthpicker */
 date: Date;
 /* function, takes in date generated via monthpicker and a string equivalent */
 onSelect: any;
}


interface MonthPickerState {
 /* the current date that monthpicker holds */
 currentDate: Date;
 /* determines if interface is visible, that is the picker itself */
 open: boolean;
 /* number corresponding to month number of current date (0 is Jan, 11 is December) */
 month: number;
 /* number corresponding to year of current date */
 year: number;
}

class MonthPicker extends React.Component <MonthPickerProps, MonthPickerState> {

        /* CLASS BODY */

}
```

5. Current Version Control Methodology

5.1. Version Control System and Continuous Integration

Prior version control information might help successors manage past development of the Credit Assessment System, check out prior versions of the system and/or even continue the version control process by expanding or optimizing the process to be specific to the needs of stakeholders of Landsbankinn. Therefore the current version control pipeline is explained.

The authors used Git for version control when developing the current version of the system, storing the code on GitHub. The authors used continuous integration, that is, all developed features and fixes were to be pushed into a branch separate from the master branch. Attempting to push changes into the master branch resulted in a rejected response from GitHub. When a feature was ready to be committed to master a pull request was created by the developer of the feature. Another team member then had to review the request before merging with the master branch to prevent undesirable violation of code rules and/or broken code. The so called Minimum Viable Product methodology was used, meaning that commits to master branch represented deployment of a new working version of the system. Therefore before setting up a new version on the master branch all project dependencies need to be available, the project needs to be buildable and all unit and integration tests need to be passed on the CI service TravisCI for continuous integration.

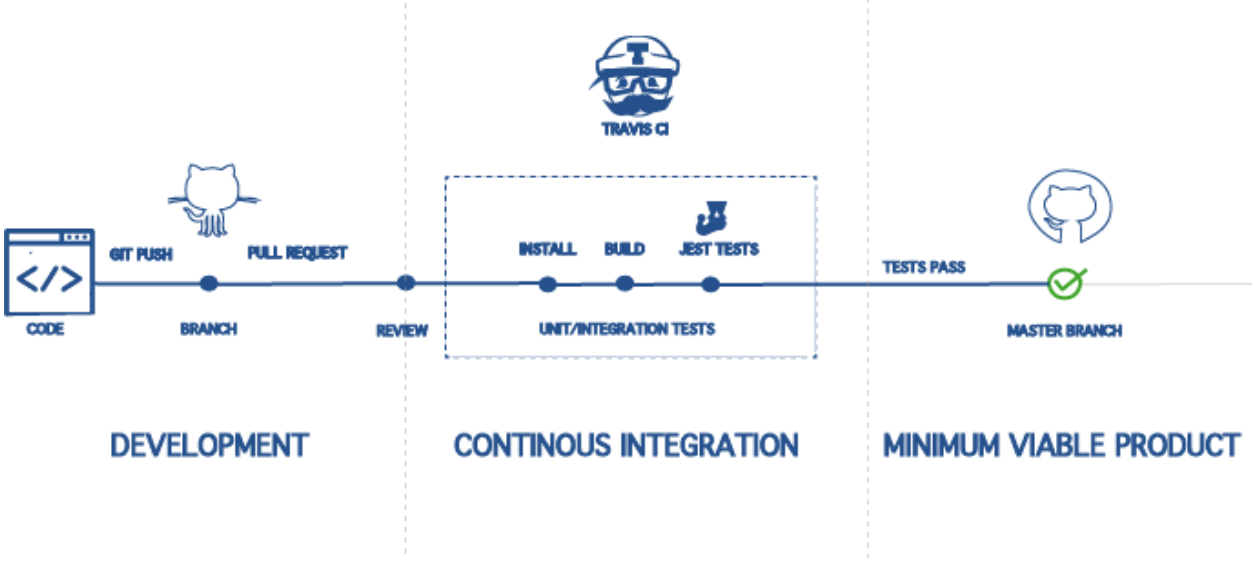


FIGURE 5.1 Current build pipeline for the Credit Assessment System.

5.2. Unit Testing

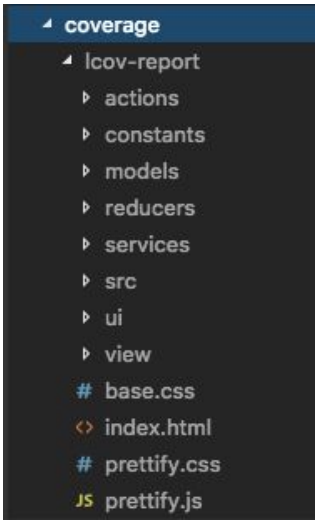
All unit tests written for the system should be placed in an appropriate subdirectory within the **__tests__** folder in the root of the project. Each component should have a single test file associated with it and each test file should only test a single component. The test file for a component should have the same relative path from the **__tests__** folder as the component has from the src folder. All tests suites have nested tests and each component tested contains up to three test suites, although most use only a single test suite.

All tests are written using Jest and are fairly self-explanatory and simple and easy to replicate. Jest is maintained by Facebook which offers a selection of useful tutorials and material. These can be found at the following page:
https://facebook.github.io/jest/docs/en/getting-started.html.

Mock data was initially created for the mock server to serve, but proved a powerful tool for testing as well. The mock data resides within the subdirectory /src/services/MockData.tsx (relative to project directory). The file contains various entries on mock assessments, assessment details and all subcategories (incomes, expenses, debts, children, etc).

In order to run unit tests for the project, one must have npm and/or yarn installed as well as NodeJS; all of which can be downloaded directly from https://nodejs.org/en/download/ or installed directly in the terminal on a linux based OS. Authors recommend using yarn although both npm and node will suffice. The following is an example command on how to run unit tests on the system from __test__ subdirectory:

```
username$ pwd
~/.../liru-credit-assessment/
username$ yarn test
```



Once this command is run, an auto-generated code coverage reports from Jest are generated in the subdirectory /src/coverage/lcov-report (*shown left*) which can be viewed component by component (by selecting html file corresponding to component in nested subdirectories) or as a whole report (via the file /src/coverage/lcov-report/index.html) in browser.

6. System's General Coding Practices

To strive for consistency future maintainers of system may wish to preserve systems coding conventions. Therefore this section documents code conventions and common practices undertaken by system authors in development of the system.

6.1. TypeScript

The following make up coding practices used in development for the Credit Assessment System specific to TypeScript and agreed upon by the system authors for the current version of the system. Most practices listed were enforced via the static checker TSLint which is configured under ./tslint.json (file path relative to project root).

NAME CONVENTIONS:

- Variable and function names are in English
- Variable and function names use the **camelCasing** convention:

```
int sumOfTwo(int firstNum, int secondNum);
```

- Global variable names use the **PascalCasing** convention:

```
int GlobalVariable;
```

- Constant names are uppercase:

```
const PI = 3.14;
```

COMMENT CONVENTIONS:

- All comments in code are written in English
- Comments spanning a single line are preceded by double slash:

```
// Comment
```

- Multi-line comments are formatted as follows:

```
/**
 *     This comment is a very long one
 *     It spans multiple lines
 */
```

CODE FORMATTING:

- Tabs are formatted as two spaces*:

```
function sumOfTwo(int x, int y) {
  return x;
}
```

* This tab settings can be configured as default when tab is pressed in Visual Studio Code. On Windows user can press F1 on their keyboard (Fn + F1 on Mac) and write in user settings in the preferences window that comes up the following command:

```
{
 "editor.insertSpaces": true,
 "editor.tabSize": 2
}
```

- Spaces are set between each operator and variable:

```
int x = x + y;
```

- Spaces are between each statement and clause:

```
if (x == y) {
  return true;                    // DO!
}
```

- Functions must be braced and must be multiline:

```
if (x == y) { return true; }      // DON'T!

if (x == y) {
  return true;                    // DO!
}
```

- Every function uses curly-brackets, even if it is only one line:

```
if (x == y)
  return true;                    // DON'T!

if (x == y) {
  return true;                    // DO!
}
```

- No spaces are set between parentheses in functions and parameters, but spaces are on the other hand set between parentheses and curly-brackets:

```
if (x == y) {
  return true;
}
```

- Opening curly-bracket is in the same line as the parameters:

```
if (x == y)
{
  return true;                // DON'T!
}

if (x == y) {
  return true;                // DO!
}
```

6.2. JSX

The following make up coding practices used in development for the Credit Assessment System specific to JSX and agreed upon by the system authors for the current version of the system:

- Each JSX tag is in its own line, regardless of size:

```
HTML:
        <div>
          <p id='paragraph'> Hello </p>
        </div>
JSX:
        <div>
          <Component/>
        </div>
```

- If a component has over than three attributes all are set in seperate lines:

```
        <Component
          attribute={...}
          anotherAttribute={...}
          attributeThree={...}
          ...
        />
```

6.3. Other General Conventions

In additions to general coding conventions the following rules were also enforced for better readability of code:

- To avoid confusion and overhead all stateless components are functions. These stateless components (often alternatively called dumb components) are arrow functions and are identified with the const keyword.

- TypeScript Interfaces are used whenever possible for their advantages of run-time error prevention.

7. System Dependencies

Table 7.1 lists all major dependencies the Credit Assessment System, the purpose of those dependencies, and nature of the version used.

DEPENDENCY	USAGE/ SPECIAL NOTES
Enzyme	Facilitates Jest unit testing by providing shallow rendering of components. A stable version used that is unlikely to be deprecated in near future.
Jest	Jest Unit testing framework developed by Facebook. Very good support and a stable version used. Unlikely to be deprecated in near future.
Kennitala Utility	Open source npm package maintained and developed by Tryggingamiðstöðin. Used to check for kennitala validity and extract date of birth from kennitala. Frequent updates and active development although version used is stable.
MaterialUI	Subset of the Material Design library developed by Google used as a front end framework for interface design. Beta version used! Since MaterialUI new major version (version 1.0) was just around the corner when development was in progress, version 1.0-beta.36 was used. However, minimal changes should be required when updating the project to 1.0 when it's released.
Moment	Formats variables in system of type date as strings. Has good support and a recent, stable version used. Unlikely to be deprecated in near future.
React Router	Handles routes of app and redirecting in system. Very good support and a stable version used. Unlikely to be deprecated in near future.
Redux	Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments, and are easy to test. Very good support and a stable version used. Unlikely to be deprecated in near future.
TSLint	Static checker for TypeScript syntax, configured by the configuration file tslint.json. Has good support and a stable version used. Unlikely to be deprecated in near future.
TypeScript	Adds support for TypeScript. Has good support and a stable version used. Unlikely to be deprecated in near future.
Webpack	Project module bundler. Has good support and a stable version used. Might be problematic when installing project using other versions of NodeJS than version 8 with is recommended by maintainers of NodeJS.

TABLE 7.1 **Major dependencies of the system, their usage and version description**

8. Capabilities and Restrictions of the Mock Server Utility

The mock server was a utility created both for unit testing and for using the system outside of the confines of Landsbankinn for the purposes of debugging, development and an important precaution measure as an option to use if a VPN connection into the Landsbankinn internal network would fail when showcasing the system to Reykjavík University. The mock server is free to use, does not serve any private or actual content and does not require authentication. The mock server does not however implement the full extent of the functionality of the API it replicates. The mock server uses data specified in a file called *MockData.tsx* (*/src/services/MockData.tsx*) and updates of that file are only done in memory, i.e. will only be stored for the current session. The following lists capabilities and restrictions of the mock server which are to beared in mind when the system is used with the mock server.

8.1. The Search Page

The search page retains all functionalities of the pre-existing API, serving mock data.

It is possible to search by customer kennitala (SSN), yielding one result, which always results in instant redirect to assessment overview page, and by name, possibly resulting in multiple individuals. The mock server is equipped with a considerable amount of mocked data for assessment details for three mock customers. Those customers are Axel Björnsson (kennitala: 170395-3369), Darri Valgarðsson (kennitala: 251295-2909) and Edda Steinunn Rúnarsdóttir (kennitala: 241095-2989). Along with those individuals are two more mock individuals; one that is registered as a bank employee (can be searched by the kennitala: 010130-2129), and another that is a minor (can be searched by the kennitala: 200600-2740). User generally does not have access to assess employees and minors and this is supported by the mock server.

8.2. The Assessment Overview Page

The search page retains all functionalities of the pre-existing API, using mock data.

The Assessment Overview page retains all its functionality when using the mock server. With the mock server it is possible to view assessments of individuals and to create a new assessment for a valid individual. However, when the new assessment is created, a redirect to its detail page is not conducted and the process is wonky although this does not affect the functionality; any created assessment will appear, can be viewed from this page and viewed for the duration of the session.

8.3. The Assessment Detail Page

The search page retains most functionalities of the pre-existing API using mock data but lacks some communication between data updates.

Any mocked assessment detail information from the mock server can be read from the assessment detail page, this gives a correct depiction of how actual data would look in the system. All operations work as they should, except printing CreditInfo documents for customers and adding same child twice to the same assessment won't result in an error.

However a major setback is that the update of one data category does not affect the other which is unfortunate since this the essence of credit assessment. To take one example, when incomes are updated, results will not update in accordance to change applicant wages as it should.

Also note that the mock server is far less tested than the actual system (that is, in terms of non-automated testing) due to the system authors having had to efficiently manage their time as resource and more on testing the system when depending on data served from the internal API.