Assessment Result Report

Candidate: Linh Trinh
Assessment: Full-Stack Engineer (NextJS/ReactJS + Kotlin/Java) Assessment
Start at: 1/14/2026 6:53:35 PM
Time taken: 121 minutes
Result: FAILED (66/120)

AI's Overview Evaluation:

The candidate demonstrates a basic understanding of full-stack software development with some foundational skills in both backend (Java/Kotlin REST API) and frontend (React) development. However, their submitted solutions contain critical errors and lack the depth, polish, and robustness expected of a senior full-stack engineer, as required by the job description.

In the backend coding test, the candidate meets most basic requirements but makes a notable error in validation logic (rejects zero-value transactions that should be allowed), and fails to implement proper error handling—critical flaws for production-grade, reliable APIs. The React coding exercise shows initial competence with state and component management but contains multiple syntactic mistakes, incorrect usage of API endpoints, and several React-specific errors (e.g. improper form submission, invalid HTML tags, and a mismatch with specified data structures). These mistakes suggest a lack of attention to detail and insufficient experience with modern React/Next.js. The theoretical answer is very superficial, omitting key areas such as proper security (authentication, HTTPS), practical data validation strategies, and in-depth discussion of data flow and error handling—well below the level expected even of a mid-level engineer.

Overall, while the candidate shows promise and the ability to write basic full-stack applications, their current skill level does not meet the high expectations for a Senior Software Engineer able to deliver production-quality work across both Next.js and Kotlin/Java, especially in a lean, product-focused startup environment. Major improvements in best practices, error handling, API correctness, frontend robustness, and architectural understanding are required for suitability to this role.

Section 1: Backend Engineering with Kotlin/Java (50 min)

Question 1.1 — code implementation

You are tasked with implementing a simple RESTful API for managing transaction records using either Kotlin or Java. Your API should support the following features:

- Creating a new transaction (via POST request).
- Retrieving a list of all existing transactions (via GET request).

Each transaction entity should have the following fields:
- `id`: A unique identifier for the transaction. This should be generated by the system when a transaction is created.
- `amount`: A non-negative decimal value representing the transaction amount.
- `description`: A non-empty string describing the transaction.
- `timestamp`: The date and time when the transaction was created (should be set by the system at creation).

Requirements:

1. Use only in-memory storage for the transactions; you do NOT need to use any database. The storage only needs to persist while the application is running.
2. Implement input validation as follows:
- `amount` must be non-negative.
- `description` must be present and not blank.
3. Expose two endpoints:
- `POST /transactions` — Accepts input data for `amount` and `description` as JSON, validates the data, creates a new transaction, and returns the created transaction (including generated `id` and `timestamp`).
- `GET /transactions` — Returns a list of all stored transactions.
4. Use either Kotlin or Java. You can use **Spring Boot / Ktor** and a JSON lib (Jackson/Kotlinx serialization).
5. You do NOT need to implement authentication, persistence beyond in-memory, or full server setup. Focus your code on the relevant logic: data structures, input validation, transaction creation, and request handling for the specified endpoints.

Assumptions:
- You can assume that the server will run as a single process and that concurrent writes are not a concern for this question's scope.
- You may provide method stubs or handler code snippets that illustrate how the endpoint logic would be structured, without setting up a complete standalone server.

Performance and Evaluation Criteria for Candidates:
- Correctness: Implements the required features and validation accurately.
- Code clarity: Code is clear, concise, and easy to understand.
- Use of appropriate data structures for in-memory transaction storage.
- Input validation logic is clearly handled and errors are reported appropriately (returning error responses when required).

Sample (example only) JSON input for POST request:
```
{
"amount": 100.50,
"description": "Payment for order #12345"
}
```

Instructions:
- Implement all required classes, methods, and handlers necessary to demonstrate the full logic for both endpoints.
- Your code should include transaction structure, in-memory storage, input validation, and HTTP request handling for POST and GET requests.
- Do NOT include any diagrams, images, or non-text content in your response.
- Do NOT provide any server bootstrapping code beyond what is needed to demonstrate the endpoint logic.

Candidate's Answer:

Example.js

```
package test.assignment.Inspius;

import lombok.Getter;
import lombok.Setter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;
import org.springframework.web.bind.annotation.*;

```

```
import java.math.BigDecimal;
import java.time.Instant;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

@SpringBootApplication
public class _1_test {

 public static void main(String[] args) {
 SpringApplication.run(_1_test.class, args);
}

@RestController
@RequestMapping("/api/v1")
 static class TransactionController {

 private final TransactionService transactionService;

 TransactionController(TransactionService transactionService) {
 this.transactionService = transactionService;
}

@GetMapping("/transactions")
 public List<Transaction> getAllTransactions() {
 return transactionService.getAllTransactions();
}

@PostMapping("/transactions")
 public Transaction createTransaction(@RequestBody TransactionRequest transaction-
Req) {
 if (transactionReq.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
 throw new IllegalArgumentException("Amount must be greater than zero");
}

 if (StringUtils.isEmpty(transactionReq.getDescription())) {
 throw new IllegalArgumentException("Description cannot be empty");
}

 var transaction = convertToTransaction(transactionReq);

```

```
    return transactionService.createTransaction(transaction);
}

    Transaction convertToTransaction(TransactionRequest transactionRequest) {
    Transaction transaction = new Transaction();
    transaction.setAmount(transactionRequest.getAmount());
    transaction.setDescription(transactionRequest.getDescription());
    return transaction;
}

}

    interface TransactionService {
    List<Transaction> getAllTransactions();

    Transaction createTransaction(Transaction transaction);
}

@Service
    static class TransactionServiceImpl implements TransactionService {

    List<Transaction> transactions = new ArrayList<>();

@Override
    public List<Transaction> getAllTransactions() {
    return transactions;
}

@Override
    public Transaction createTransaction(Transaction transaction) {
    transaction.setId(UUID.randomUUID().toString());
    transaction.setTimestamp(Instant.now());
    transactions.add(transaction);
    return transaction;
}
}

@Getter
@Setter
    static class TransactionRequest {
```

```
90    private BigDecimal amount;
91    private String description;
92   }
93
94   @Getter
95   @Setter
96    static class Transaction {
97    private String id;
98    private BigDecimal amount;
99    private String description;
100   private Instant timestamp;
101  }
102
103
104  }
```

Section 2: Frontend Engineering with React/NextJS (50 min)

Question 2.1 — code implementation

You are tasked with developing a minimal user interface in ReactJS. You may use NextJS if you are comfortable with it (this will be considered for bonus points, but is not required). The interface should interact with an external API to display a list of transactions and allow users to add new transactions. Follow the requirements below:

API Requirements:
- The transaction API endpoint is: `https://api.example.com/transactions`.
- A GET request to this endpoint returns a JSON array of transaction objects, each with the following structure:

```json
[
{
"id": 1,
"date": "2024-05-01",
"description": "Book purchase",
"amount": -15.50
},
{
"id": 2,
"date": "2024-05-02",
"description": "Coffee Shop",
"amount": -3.75
},
{
"id": 3,
"date": "2024-05-03",
"description": "Salary",
"amount": 1200.00
}
]
```

- To add a new transaction, send a POST request with a JSON body containing `date` (string), `description` (string), and `amount` (number). Example JSON body:

```json
{
"date": "2024-06-01",
"description": "Groceries",
"amount": -54.95
}
```
- Assume all API responses are successful and immediate for this exercise.

User Interface Requirements:
1. When the application loads, fetch the transactions from the API and display them in a table. Each row should display the transaction's `date`, `description`, and `amount`.
2. Below the transaction table, provide a form that allows the user to add a new transaction. The form should collect `date`, `description`, and `amount` values.
3. Upon submitting the form, a POST request should be sent to the API endpoint to add the new transaction. After a successful submission, the interface should update to show the new transaction in the table.
4. Organize your code using functional React components. Organize responsibilities with at least two custom components (for example, a component for the table and a component for the form).
5. Use React's built-in state and effect hooks. Do not use any external state management or form libraries.
6. Basic styling (e.g., using plain CSS or inline styles) is acceptable but not required.
7. You are not required to handle API errors, loading states, or authentication for this exercise.

Assumptions:
- The API endpoint is available and works as described above.
- You may use fetch or any standard browser HTTP API for network requests.
- You may structure your files as you see fit, but only provide code snippets that are necessary to demonstrate your solution within a single file (e.g., a single component file or page).

Evaluation Criteria:
- Correctness of data fetching, form submission, and state updates.
- Proper use of React component composition and hooks.
- Clarity, readability, and organization of code.
- Adherence to requirements above.
- (Bonus) If you use NextJS, use the page or component file layout consistent with NextJS conventions.

Candidate's Answer:

TransactionList.js

```
function TransactionList({ transactions }) {
 return (
<div className="transaction-list">
<h2>TransactionList</h2>

{
 transactions.length == 0 ? (
<p>No transactions found </p>
) : (
<table>
<thread>
<tr>
<th>Date</th>
<th>Description</th>
<th>Amount</th>
```

```
</tr>
</thread>
<tbody>
{
 transactions.map((transaction) => (
<tr key={transaction.id}>
<td>{transaction.date}</td>
<td>{transaction.amount}</td>
<td>{transaction.description}</td>
</tr>
))
}
</tbody>
</table>
)
}
</div>
)
}

export default TransactionList;
```

AddTransaction.js

```
import React, { useState } from "react";

function AddTransaction({ onAddTransaction }) {
 const [date, setDate] = useState('');
 const [amount, setAmount] = useState('');
 const [description, setDescription] = useState('');

 const handleSubmit = (e) => {

// validate
 if (!date || !description || !amount) {
//log
 return;
}
// create new trans

```

```
    const newTransaction = {
    date, description, amount: parseFloat(amount)
  }

    onAddTransaction(newTransaction);

  // reset
    setDate('');
    setDescription('');
    setAmount('');

  }

    return (
  <div className="create-transaction">
  <h2>Add transaction</h2>
  <form onSubmit={handleSubmit}>
  <div className="form-group">
  <label for="start">Start date:</label>
  <input
    type="date"
    id="date"
    value={date}
    onChange={(e) => setDate(e.target.value)}
    required
  />

  </div>

  <div className="form-group">
  <label for="start">Description:</label>
  <input
    type="text"
    id="description"
    value={description}
    onChange={(e) => setDescription(e.target.value)}
    placeholder="Enter your desc"
    required
  />
  </div>
```

```
56
57  <div className="form-group">
58  <label for="start">Amount:</label>
59  <input
60   type="number"
61   id="amount"
62   value={amount}
63   onChange={(e) => setAmount(e.target.value)}
64   placeholder="0.00"
65   required
66  />
67  </div>
68
69  <button type="submit" className="btn-submit">
70   Add Transaction
71  </button>
72  </form>
73  </div>
74  )
75  }
76
77
78  export default AddTransaction;
```

App.js

```
1   import './App.css';
2   import { useEffect, useState } from 'react';
3   import AddTransaction from './components/AddTransaction';
4   import TransactionList from './components/TransactionList';
5
6   function App() {
7
8    const [transactions, setTransactions] = useState([]);
9    const [loading, setLoading] = useState(false);
10   const [error, setError] = useState('');
11
12   useEffect(() => {
13   fetchTransactions();
14
```

```
}, []);

 const fetchTransactions = async () => {
 setLoading(true);
 try {
 const response = await fetch('http://localhost:8080/api/v1/transactions/entire');
 if (!response.ok()) {
//log error
 return;
 }

 const data = await response.json();

 setTransactions(data);
 } catch (e) {
// throw error
 setError(`Fail to load transactions: ${e.message}`)
 } finally {
 setLoading(false);
 }
 }

 const addTransaction = async (newTransaction) => {
 setLoading(true);
 try {
// call api
 const response = await fetch('http://localhost:8080/api/v1/transactions', {
 method: 'post',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify(newTransaction)
 });

 setTransactions(newTransaction, ...transactions);

 if (!response.ok()) {
//log error
 return;
 }

 const data = await response.json();
```

```
54
55    setTransactions(data);
56   } catch {
57   // throw error
58    setError(`Fail to add transaction: ${e.message}`)
59   } finally {
60    setLoading(false);
61   }
62
63
64   }
65
66    return (
67   <div className="App">
68   <h2>Transaction manager</h2>
69
70   {error && <div>Error {error} </div>}
71
72   <AddTransaction onAddTransaction={addTransaction}></AddTransaction>
73
74   {loading ? <div>Loading transactions ... </div> :
75   <TransactionList transactions={transactions}></TransactionList>
76   }
77
78   </div>
79   );
80   }
81
82   export default App;
```

Section 3: Full-Stack Integration & Applied Problem Solving (20 min)

Question 3.1 — theoretical discussion

Describe in detail how you would integrate a frontend application with the backend system you have previously implemented. Your answer should address the following points:

1. **API Route Structure**
- Explain how you would design and structure the API endpoints to support the required features and workflows. Mention how you would organize your routes (e.g., RESTful patterns, route naming conventions, versioning).

2. **Data Flow Management**
- Describe how data would move between the frontend and backend. Specify how you would send requests from the frontend to the backend, how the backend would process and respond to those requests, and how the frontend would handle responses.

3. **Loading and Error States**
- Discuss strategies you would use in the frontend to manage loading states (e.g., during requests) and error states (e.g., validation errors, network errors, or server errors). Explain how you would communicate these states to the user.

4. **Security Considerations**
- Identify at least two key security measures you would implement to protect API endpoints and data exchanged between frontend and backend (e.g., authentication, authorization, HTTPS, CORS handling).

5. **Data Validation**
- Explain where and how you would validate data: on the frontend, the backend, or both? Outline potential approaches and reasoning for your choices (e.g., user input validation, server-side checks).

Be specific in your explanations, clearly discuss your reasoning, and make sure your answer addresses all of the above points. Do not include any diagrams or images in your response, and assume only standard, text-based technologies available in most programming environments (e.g., HTTP, REST, JSON).

Candidate's Answer:

Example.js

```
1. API Route Structure
 I design restful endpoint base on method rule:
- Get /transactions for get all transactions
- Post /transactions for create new transaction
- Put /transactions/{id} for update transaction
- Delete /transactions for delete transaction
...
 I using prefix api/v1/... for versioning

2. Data Flow Management
render ui → user_action → → components(eg. onClick,..)) →  event handle (eg onclick)
→ api service (fetching method) → http request → backend server(controller, service, repos-
itory) → database → http response → update state → re-render ui → user see result

3. Loading and Error States
- I use useState to manage state for simple and ease to read, efficent for small application
- I will use 1 state to store error, we need display error for user know how their action will effect
to application

4. Security Considerations
- Use CORS handling for prevent cross origin access

5. Data Validation
```

[redacted]