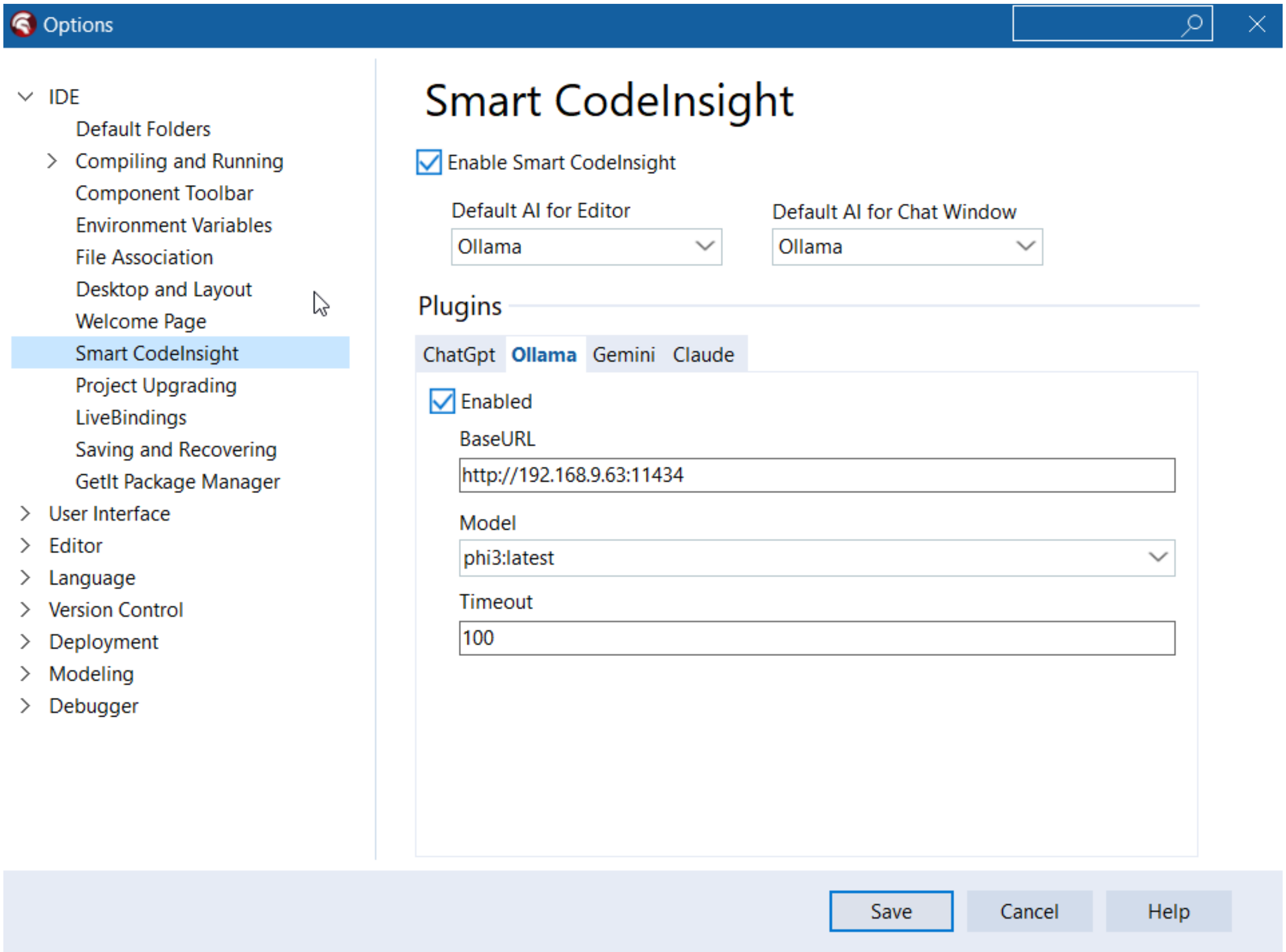
Microcredencial Implementación y uso de IA generativa en Delphi

Tarea Vibe Coding

Xavier López Pujol

Configuración

Usando Ollama y el modelo Phi3. Ollama está corriendo en otra máquina que no es la de desarrollo de delphi.
Esta es la configuración realizada:



Smart CodeInsight

☑ Enable Smart CodeInsight

Default AI for Editor: Ollama

Default AI for Chat Window: Ollama

☑ Enabled

BaseURL: http://192.168.9.63:11434

Model: phi3:latest

Timeout: 100

(poner timeout a 0 si da error)

Hay que invocar a ollama que esté en escucha en cualquier ip cuando se accede a través de lan:

```
$env:OLLAMA_HOST = "0.0.0.0" ; ollama serve
```

Creación de código

La creación de código se ha hecho desde Claude, ya que el código que retornaba Ollama tanto con deepseek como phi3 tenía demasiados inexactitudes. El prompt especificado es el siguiente:

in delphi 12 create a class to get suggestion list words from a list of strings from a suggested string

Prompt para crear unit tests:

create unit test for this class following unit tests library from mormot

Prompt para corregir el código generado:

the unit test with mormot are not correct. please apply correction to that. for example checknotnull doesn't exists

Uso de opción Unit Search (AI)

Debido a que Claude no proporcionó el uses adecuado, al buscar sobre TComparer

```
(************* Find Unit(AI) ***************
System.Generics.Collections;
*)
```

Aunque el resultado no es correcto y finalmente claude dijo que está en System.Generics.Defaults

Corrección de bugs

La porción de código que se ha introducido el error (se ha quitado el .free de la lista) es la siguiente

```
function TWordSuggestion.GetSuggestionsWithScore(const AInput: string):
TArray<TSuggestionMatch>;
var
  SuggestionList: TList<TSuggestionMatch>;
  I: Integer;
  Score: Double;
  Match: TSuggestionMatch;
  NormInput: string;
begin
  SuggestionList := TList<TSuggestionMatch>.Create;
  try
    NormInput := NormalizeString(AInput);

    // Skip if input is too short
    if Length(NormInput) < FMinMatchLength then
    begin
      SetLength(Result, 0);
      Exit;
    end;
    // Calculate scores for all words
    for I := 0 to FWordList.Count - 1 do
    begin
      Score := CalculateSimilarityScore(AInput, FWordList[I]);

      if Score > 0 then
      begin
        Match.Text := FWordList[I];
        Match.Score := Score;
        Match.MatchType := GetMatchType(AInput, FWordList[I]);
        SuggestionList.Add(Match);
      end;
    end;

    // Sort by score (descending)
    SuggestionList.Sort(TComparer<TSuggestionMatch>.Construct(
      function(const Left, Right: TSuggestionMatch): Integer
      begin
        Result := CompareValue(Right.Score, Left.Score); // Descending
      end));

    // Return top suggestions
    SetLength(Result, Min(SuggestionList.Count, FMaxSuggestions));
    for I := 0 to High(Result) do
      Result[I] := SuggestionList[I];

  finally
      // se ha quitado el SuggestionList.Free;
  end;
end;
```

El resultado que ha dado usando Phi3 es el siguiente:

The Delphi code provided seems mostly fine, but there are some potential issues that could cause bugs or unexpected behavior. Here are a few possible concerns:

1. **Variable Initialization Issue - Before using the `Result` variable as an array in this method signature `function TWordSuggestion.GetSuggestionsWithScore(const AInput: string): TArray<TSuggestionMatch>`, it should be initialized to ensure that you don't get errors when attempting to fill it with elements, especially since Delphi does not have a built-in array type like in C# or Java. However, the code provided uses `SetLength(Result, 0)`, which is correct and initializes an empty TArray of TSuggestionMatch objects:**

```
function
TWordSuggestion.GetSuggestionsWithScore(const AInput: string):
TArray<TSuggestionMatch>;
```

```
var
  SuggestionList: TList<TSuggestionMatch> = nil;
// Initialize empty list here instead of creating it inside the try block, but
this can be omitted in Delphi since we're assigning to Result directly after
initializing.
begin
```

2. **Unassigned Exception** - In case `CalculateSimilarityScore` or any other method called within returns an error (e.g., NullReferenceException), it could crash the application without proper exception handling:

Consider wrapping risky operations in try-except blocks to catch and handle these exceptions gracefully, e.g.:

```
for I := 0 to
FWordList.Count - 1 do begin
  Score := CalculateSimilarityScore(AInput,
FWordList[I]); // Possible NullReferenceException here if the list is empty or
some objects are removed at runtime without updating `FWordList` count. It's
best practice not to rely on exceptions as a normal flow of control in Delphi
but handle it properly:
  try
    if Score > 0 then begin
      // Code
continues unchanged...
```

3. **MatchType Calculation Issue** - If the `GetMatchType` method can return different types and they're not all compatible with your code, you could encounter issues or unexpected behavior (e.g., trying to access properties of an invalid type):

```
for I := 0 to
FWordList.Count - 1 do begin
    Score := CalculateSimilarityScore(AInput,
FWordList[I]); // Possible NullReferenceException here if the list is empty or
some objects are removed at runtime without updating `FWordList` count. It's
best practice not to rely on exceptions as a normal flow of control in Delphi
but handle it properly:
    try
      Match := TSuggestionMatch.Create; //
Make sure all properties match the expected type and create an instance before
using them here, otherwise you might encounter run-time errors when accessing
uninitialized fields or methods on nil objects for example:
        if Score >
0 then begin
          Match.Text := FWordList[I];

SetMatchType(AInput, Match); // Consider initializing `SetMatchType` with proper
arguments to avoid potential run-time errors (e.g., trying to access properties
on a nil object or passing invalid parameters). This could be done as:
```

```
Score := CalculateSimilarityScore(NormInput, FWordList[I]); // Consider moving
the NormalizeString outside of 'try' since it will never throw exceptions and
can potentially return NULL which you would have to check for before using its
value. It might also be a good idea not to assign an intermediate result
directly here if this function could possibly fail or cause issues, but that
depends on how `NormalizeString` works:
              NormalizedInput :=
NormalizeString(NormInput); // Consider handling potential
NullReferenceExceptions from calling `NormalizeString`:
                Score
:= CalculateSimilarityScore(AInput, FWordList[I]); // Again consider using
try-except blocks or checking for NULL values before passing them to functions
that could throw exceptions. In this case we don't expect any but it is good
practice:                        if not Assigned(NormalizedInput) then exit;

Match := TSuggestionMatch.Create;
          Score := // Consider moving the
score calculation outside of 'try', since you can assume that
`CalculateSimilarityScore` doesn't throw an exception and using it as-is here:
SetMatchType(NormalizedInput, FWordList[I]);// Same considerations for handling
exceptions from this method call
          SuggestionList.Add(Match);

end; // No need to exit if score is <= 0 since we are collecting all scores > 0
in a TArray which will not be empty:
    finally
      Match.Free; // Properly
free any allocated memory or unassigned objects here instead of just relying on
the garbage collection mechanism (Delphi's GC could potentially lead to leaks if
we forget to do it):                       SuggestionList.Clear();
```

4. **Resource Management Issue** - The code does not explicitly release resources in case of exceptions or when no suggestions are found: Delphi has a `try...except` block, but there's also the use of Finalization which can be used to ensure that any object created within this try-block is freed even if an exception occurs. However, it should generally not replace proper error handling and freeing memory in 'finally':

```
 finally
    SuggestionList.Free; // Ensure
all resources are properly disposed of:
      FreeAndNil(SuggestionList);

end;
end;
```

5. **MatchType Logic Issue** - If `GetMatchType` is not defined as a standalone method with proper logic, there's no way to know if the match type will always return expected values (e.g., what

happens when it encounters an empty input or unexpected characters). This can lead to bugs:

```
function GetMatchType(const AInput: string; const MatchText:
string): TStringListEnumType; // Assume this function returns some type of
enumeration that represents the match types (e.g., typo, exact match, etc.) Here
we need proper documentation or unit tests to ensure its logic works as expected
under different conditions
```

Overall, while there's no immediate crash in your code snippet without actual testing and implementation details for methods like `NormalizeString`, `CalculateSimilarityScore`, and `GetMatchType`, these concerns are worth considering. It would be helpful to have more context about the expected behavior of those missing parts or specific cases where you expect bugs, so that we can provide better insights into potential issues related to them in your code.

Dando en el punto 4 con el bug introducido!

Este es el resultado usando API Chatgpt5, ya que con deepseek, la máquina que tengo no tiene GPU y tarda muchísimo el resultado.

Possible bugs and improvements in the code:

- Memory leak (major)
    - SuggestionList is created but never freed. The finally block is empty, so the list leaks.
    - Fix: Free the list in the finally block (or use FreeAndNil).
      Example: finally

SuggestionList.Free; end;

• Inconsistent use of normalization

NormInput is computed but only used for the length check. The scoring and match type use AInput, which may be unnormalized.

- If normalization is intended for scoring/matching, pass NormInput to CalculateSimilarityScore and GetMatchType (and consider normalizing FWordList[i] similarly or presuming words are already normalized).
  Example change: Score := CalculateSimilarityScore(NormInput, FWordList[I]); Match.Text := FWordList[I]; Match.Score := Score; Match.MatchType := GetMatchType(NormInput, FWordList[I]);
- Reuse of a single local record variable
    - Match is defined once outside the loop and reused. While likely fine for a record, it's safer to construct a fresh record per iteration (to avoid carrying over any uninitialized fields if TSuggestionMatch later gains more fields).
      Example: var S: TSuggestionMatch; S.Text := FWordList[I]; S.Score := Score; S.MatchType := GetMatchType(NormInput, FWordList[I]); SuggestionList.Add(S);
- Min/SetLength edge cases
    - If FMaxSuggestions could be negative, Min(...) might yield a negative length, causing runtime errors. Ensure FMaxSuggestions is non-negative (or guard against it).

- Minor performance/consistency improvements
    - If you normalize for scoring, consider also normalizing the word being compared (or clearly document whether FWordList is already normalized).
    - The current top-N extraction logic is fine, but you could simplify/clarify by explicitly computing Count := Min(SuggestionList.Count, FMaxSuggestions) and then SetLength(Result, Count).
- Compile-time dependencies
    - If you use Min, ensure you have the appropriate unit (Math) in uses. Some Delphi projects require Math.Min for integers; in others Min is available via System. If not present, you'll get a compile error.

Recommended corrected, self-contained sketch:

function TWordSuggestion.GetSuggestionsWithScore(const AInput: string): TArray; var SuggestionList: TList; I: Integer; Score: Double; NormInput: string; S: TSuggestionMatch;

Count: Integer; begin SuggestionList := TList.Create;

try NormInput := NormalizeString(AInput);

`// Skip if input is too`

short if Length(NormInput) < FMinMatchLength then begin

SetLength(Result, 0); Exit; end;

`// Calculate scores for all`

words for I := 0 to FWordList.Count - 1 do begin Score := CalculateSimilarityScore(NormInput, FWordList[I]); if Score > 0 then

begin S.Text := FWordList[I]; S.Score := Score;

S.MatchType := GetMatchType(NormInput, FWordList[I]);

SuggestionList.Add(S); end; end;

`// Sort by score`

(descending) SuggestionList.Sort(TComparer.Construct(

function(const Left, Right: TSuggestionMatch): Integer begin

Result := CompareValue(Right.Score, Left.Score); end));

`// Return`

top suggestions Count := Min(SuggestionList.Count, FMaxSuggestions);

SetLength(Result, Count); for I := 0 to Count - 1 do Result[I] := SuggestionList[I];

```
finally SuggestionList.Free; end; end;
```

In short: the biggest flaw is the memory leak from not freeing the SuggestionList. Also consider using normalized input for scoring/match type and initialize per-iteration record data for safety.